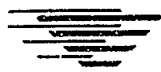AD-A235 511

Carnegie-Mellon University
Software Engineering Institute

Support Materials for

Software Configuration Management

Support Materials SEI-SM-4-1.0

DTIC
ELECTE
MAY 2 3 1991
S
C
D

91-00325

Support Materials
for

Software Configuration Management

SEI Support Materials SEI-SM-4-1.0

September, 1986

Edited by

James E. Tomayko
The Wichita State University

Carnegie-Mellon University
Software Engineering Institute

This technical report was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official
DoD position. It is published in the interest of scientific and technical
information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

JOHN S. HERMAN, Capt, USAF
SEI Joint Program Office

Contents

A Configuration Management Example

James E.Tomayko
The Wichita State University

This section contains examples of discrepancy reports, change requests, and requests for enhancements to products to show their differences and similarities.

Real-Time Hacking, Inc. (RTH), a defense contractor, decided to write the control software for the elevator in its new engineering building to provide its programmers some experience with Ada before its use was mandated in new products. A team leader prepared the following specification and partial design of the program:[1]

Functional Specification

The program is a procedure RUN_ELEVATOR that controls an elevator serving 8 floors of a building, numbered from 1 to 8 (no basement).

At each floor in the building are two elevator call buttons-UP and DOWN (except for the first floor which does not have a DOWN button and the top floor which does not have an UP button). Inside the elevator there are 8 FLOOR buttons, one for each of the 8 floors and an OPEN button. FLOOR button marked I is depressed by a passenger to get off at floor I and the OPEN button is depressed to prolong the period the elevator door is open.

Elevator Specification: The elevator car behaves as follows:

1. It services the 8 floors carrying, passengers up and down: Its *home* floor is the first floor (the building lobby). Whenever there are no requests for use, it stations itself at the home floor.

2. When going up, the elevator services all requests for stops on floors above its current position; similarly when it is going down. The elevator tries to minimize the number of changes in direction: (No person waits forever.)

3. The elevator opens its door for 5 seconds: Every time the OPEN button is pressed, the door is kept open for one extra second. However, pressing the OPEN button when the door is closed has no effect.

Physical Details of the Elevator: Depressing an elevator button causes a hardware interrupt (with a possible parameter) on the computer associated with the elevator: These interrupts are queued automatically: Hardware addresses corresponding to these interrupts are

Button	Address	Function
DOWN(I)	8#1000#	Request to go down from floor I
UP(I)	8#1010#	Request to go up from floor I
FLOOR(I)	8#1020#	Stop at floor I
OPEN	8#1030#	Delay closing door by one second

A package ELEVATOR with the following specification is available.

---

[1]The specification and partial implementation of this example are adapted by permission from Narain Gehani, *Ada: An Advanced Introduction* (Prentice-Hall, 1983, pp. 181-200). Ada is a trademark of the U.S. Department of Defense.

```
package ELEVATOR is
    procedure MOVE_UP_ONE_FLOOR;
    procedure MOVE_DOWN_ONE_FLOOR;
    procedure CLOSE_DOOR;
    procedure OPEN_DOOR;
end ELEVATOR;
```

Elevator Movement Timing Characteristics: The elevator movement consists of three phases-the car first accelerates to steady speed, then travels at steady speed and, finally, decelerates to a stop. The elevator takes 1.80 seconds to go from a stationary position at floor I to a stationary position at floor I+1 (the characteristics are the same whether the elevator is going up or down)-0.40 seconds to accelerate to steady speed while covering the distance A_1B_1, 1.00 seconds traveling at steady speed to cover the distance B_1C_1, and 0.40 seconds decelerating to a stop while covering the distance C_1D_1. A_1B_1 is equal to C_1D_1, A_1 coincides with D_1-1 and D_1 coincides with A_1+1.

If there is no need for the elevator to stop at the next floor then it must be given another move command before it starts decelerating, i.e., at or before position C_1. There are two cases:

1. Suppose the elevator is in a stationary position at the time the first move command is given: Then the elevator should be given the next move command at most 1.40 seconds after the previous move command

2. Suppose the elevator starts from floor 1-1 or earlier. It does not stop at floor 1 and is not to stop at floor I+1 either. It was last instructed to keep moving at position C_1-1. It must now be instructed to keep moving at C_1. Traveling at steady speed the elevator covers the distance A_1B_1 or C_1D_1 in half the time it takes when accelerating or decelerating. Consequently, it covers the distance $C1$-$1C_1$ in 1.40 seconds. The next move command, as in the first case, must be given at most 1.40 seconds after the previous move command.

Design

Requests for elevator service, to go up or down, or to get off, are accepted by a task REQUEST_DB (requests data base), which also keeps track of these requests. Task ELEVATOR_CONTROL controls the elevator using commands provided in the package ELEVATOR: It also accepts requests from passengers, made by depressing the OPEN button, to keep the elevator door open longer than the normal period. Task ELEVATOR_CONTROL interacts with the task REQUEST_DB to

1. determine the next elevator destination based on pending requests for elevator serv_ce, and

2. supply information specifying the floors that have been serviced.

The interaction between tasks ELEVATOR_CONTROL, REQUEST_DB, package ELEVATOR and the elevator itself is illustrated in Figure 1.

At any time, the elevator will be in one of three states-UP, DOWN or NEUTRAL: States UP and DOWN indicate that the elevator is going in the direction implied by its state in response to passenger requests. The NEUTRAL state indicates that the elevator is not responding to a request but that it might be headed toward its home floor if it is not already there.

The Elevator System

elevator car

REQUEST DB

UP,   DOWN, FLR

OPEN

ELEVATOR CONTROL

control signals

ELEVATOR

○ task

☐ package
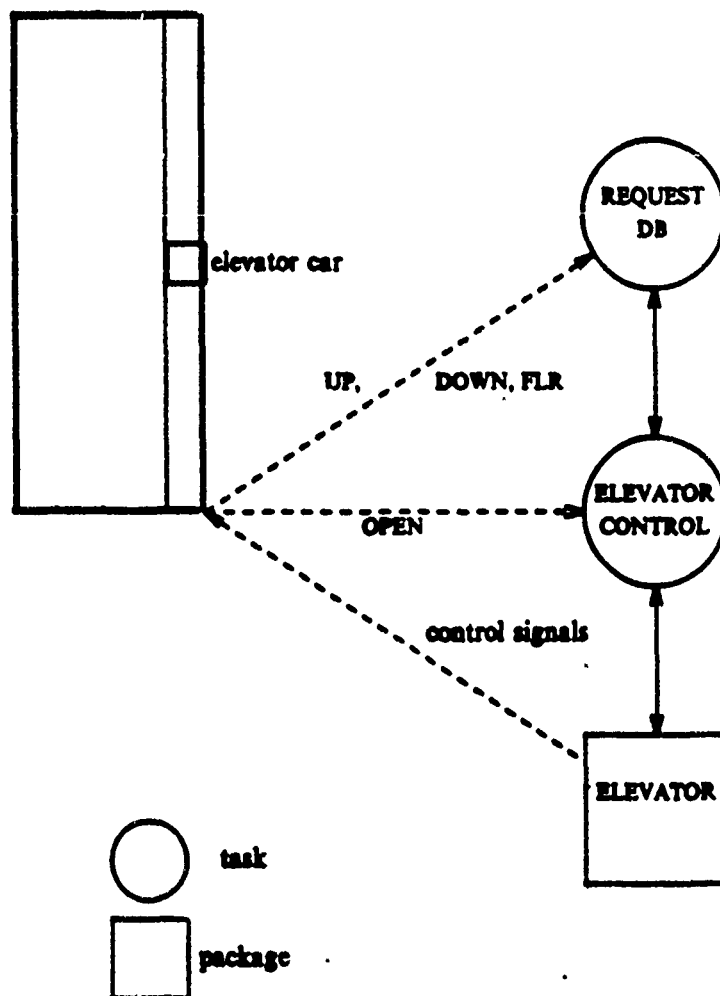
Figure 1

Some constant and type declarations used in the implementation are:

```
HOME: constant:= 1;
N: constant:= 8; --number of floors

subtype STORIES is INTEGER range 1..N;
type STATE is (UP, DOWN, NEUTRAL);

NORMAL_OPEN_TIME: constant DURATION:= 5.0;
EXTRA_OPEN_TIME: constant DURATION:= 1.0;
NEXT_MOVE_TIME: constant DURATION:= 1.39
        --the next move command must be given at most
        --1:40 seconds after the previous move command;
        --selection of 1:39 is arbitrary except for the
        --above constraint
```

The specification of task ELEVATOR_CONTROL is

```
task ELEVATOR_CONTROL is
        entry OPEN; --keep the door open one second longer
        --associate hardware interrupt location with entry OPEN
         for OPEN use at 8#1030#;
end ELEVATOR_CONTROL;
```

The specification of task REQUEST_DB is

```
task REQUEST_DB is
        entry DEST(CUR_STATE: in STATE; CUR_FLOOR: in STORIES;
                NEW_STATE: out STATE;
                NEW_FLOOR: out STORIES);
            --computes the new destination and direction
            --based on the current location, current
            --direction and pending requests
        entry REQUESTS (B: out BOOLEAN);,
            --TRUE returned in B if there is any pending request
            --for elevator service and FALSE otherwise
        entry CLEAR_GO(DIR: in STATE; I: in STORIES);
            --picked up passenger(s) going up or down from floor I
        entry CLEAR_OFF(I: in STORIES);
            --passenger(s) let off at floor I

        --the following entries correspond to passenger
        --requests for elevator service
            entry DOWN (I: in STORIES);
            entry UP(I: in STORIES);
            entry FLOOR(I: in STORIES):,

        --associate hardware interrupt locations with entries
            for DOWN use at 8#1000#;
            for UP use at 8#1010#;
            for FLOOR use at 8#1020#;
end REQUEST_DB;
```

RTH's configuration manager placed the Functional Specification and Design documents under configuration control when the program entered its implementation phase, even though it was an informal development project and not intended as a product. She also wanted to get some Ada experience. This turned out to be fortunate, as a flurry of discrepancy reports began to surface during integration testing. A configuration control board consisting of the author of the functional specification, a representative from the test and evaluation team, the division manager whose engineers were doing the Ada excercise, and the configuration manager hastily convened. The division chief had the responsibility for the final decision on discrepancy reports and change requests.

A Discrepancy Report

One of the discrepancy reports the board considered was turned ir _y a test engineer (See Figure 2).

The origins of this problem lie in a typographical error that mushroomed across the system. The existing mechanical elevator control system accelerated and decelerated in 1.4 seconds. The author of the Functional Requirements mistakenly wrote 0.4 seconds. The engineers designing the changes to the hardware needed to transition to the digital system, aware of the Functional Specification of the software, interpreted the value 0.4 as a virtual change request to their equipment. Since the work was being done by RTH's in-house staff and not by the elevator manufacturer, they did not realize that 0.4 seconds was a little fast, they thought it was a benefit to be gained by using software control! The engineer writing the discrepancy report, thinking that the software was the only new element in the system, flagged the software, but put a few question marks next to the hardware classification because he really did not know.

The configuration control board quickly realized that the software was not controlling acceleration and deceleration rates, but merely had to be aware of them for the NEXT_MOVE_TIME parameter to be correct. Therefore an easy fix to the documentation and code could be made, but it took nearly a week to run down who hau changed the hardware. If the system had been organized as a product development effort from the start, a design review on the hardware side should have detected the error in changing the elevator's existing acceleration and deceleration rates.

Note that the discrepancy report form used here has provision for tracking the disposition all the way through to the verification of the actual changes, including forcing the originator to sign off that the analysis of the problem is understood by him.

A Change Request

During the testing period of the new system, a handicapped secretary sitting in a wheelchair found herself in the back of the elevator with several people standing in front of her. When she reached her floor, the standees filed out somewhat slowly. She got near the door as it started to close, hit the 'Door Open' button once, and then nearly got trapped when she could not clear the doors during the one second delay. The result of this incident was Figure 3, a change request.

The configuration control board evaluation of this change was fairly straightforward, in that it was obviously needed. However, the change cost considerable money as the implementation group first tried stacking 'Door Open' requests so that every time someone pushed the button a one second delay was stored. That way five pushes equalled five seconds, and the users could actually control the length of the delay. One day the president's four-year-old hit the button 36 times while the doors were open, resulting in *another* change request by the person who had just boarded the elevator wanting to go down. The original change request form used had no provision for sign-offs by the verfication or quality assurance teams, so the implementor's interpretation of the phrase 'to 3 seconds' was not challenged. The final fix was to change the value of the constant EXTRA_OPEN_TIME to 3.0 seconds.

Requests for Enhancements

After the RUN_ELEVATOR program finally was in operational use, the president of Death Rays, Inc. visited RTH. The president of RTH, made aware of the Ada elevator controller due to his son's exploit, bragged to

Death Rays' president about the program. It turned out that Death Rays was completing a new corporate headquarters, and became interested in purchasing the program to prove that they had some Ada in house. In order to help keep Death Rays' business, the president of RTH ordered a new version of the program.

The configuration manager was infuriated. What had started as an ad hoc programming project had evolved into an ongoing product development effort. The program was informally described, had no user or installation documentation, and no consistent control had been applied to its evolution. The cost of adding discipline at this point was expensive, yet no formal control board existed to draw attention to this fact. Production of the new version of the program began. The implementors thought it would be an easy deal: find out how many floors, check out the constants (this time they called the elevator manufacturer and subcontracted any hardware changes to them), and everything would be an easy fix.

The program was changed, delivered, and installed. One hour later the customer version of a discrepancy report (an Information and Assistance Request) arrived by messenger (see Figure 4). The members of the ad hoc configuration control board met hastily over lunch. The problem and its resolution can be deduced from the form.

Several months later, after everyone had nearly forgotten about RUN_ELEVATOR, a friend of the president of Death Rays called the president of RTH. He was building a huge new corporate headquarters. It had *three* elevators. The president of RTH, still wanting to keep Death Rays' business, picked up the phone to call the software shop.....

BOEING

SOFTWARE PROBLEM REPORT

☒ XXX ERROR
☒ XX

☐ IMPROVEMENT

SPR NO: _____

REF NO: _____
RVER-PRR-ECPI

PROBLEM: ORIGINATOR'S NAME __John Smith__

SYSTEM FAILING __Run_Elevator__   VERSION NO. __1.1__   FILE/MODULE NAME _____

RELATED SYSTEMS _____   RELATED TEST CASE _____

CLASSIFICATION:

☒ SOFTWARE
☐ HARDWARE   ???
☐ COMMENT/POL
☐ DOCUMENTATION
☐ INFORMATION ONLY

CORRECTION REQUIRED BY:

ANALYSIS TO BE DONE BY

PROBLEM DESCRIPTION: __The elevator we are testing using the Version 1.1 of the software accelerates and decelerates excessively fast. People have trouble staying on their feet!__

ORG _____ GROUP _____ SIGNATURE __J C Smith__ DATE __8/7/86__

ANALYSIS: (PREPARED BY RESPONSIBLE SOFTWARE DESIGN ORGANIZATION)   RECEIVED DATE __8/8/86__

EXPLANATION: __Requirements error--- the values stated in Functional Specification, section 1.3 for acceleration/ deceleration are 0.40 seconds for each. Actual values are 1.4 seconds for each.__

CLASSIFICATION:

☐ CODING
☐ DESIGN
☐ COMMENT/POL
☒ DOCUMENTATION
☐ ENVIRONMENT
☐ NO S/W CHANGE REQD
☐ REPORTED ON _____ PREVIOUSLY
☐ OTHER CONFIG AFFECTED (SPR NO. _____)

SIGNATURES: __DC__ ANALYST _____ DATE __8/11/11__ ORIGINATOR __JS__ _____ DATE __8/11/11__

CORRECTION: (BRIEF DESCRIPTION OF WORK AND LIST OF MODULES CHANGED)

SYSTEM WORK __Since acceleration and deceleration are functions of the hardware, the only change needed is in the declaration of NEXT_MOVE_TIME in the Design document, as follows:__

__NEXT MOVE TIME: constant DURATION:= 2.39__
__associated comment and Functional Spec also__ SIGNATURE __DC__ DATE __8/22/86__

DOCUMENTATION WORK: (BRIEF DESCRIPTION OF DOCUMENTATION WORK) __Functional Specification Section 1.3 changed, Design changed.__

S/W D NO.	PAGE NO.	DCN/ADCN NO.	DATE
RE-FS-001	12		
RV-FC-002	56		

CLOSURE APPROVAL SIGNATURES: _____ DATE __8/24/86__

CONFIRMATION: (CHANGE VERIFIED BY S.C.M.)

STM NO. _____

AVAILABLE IN VERSION __1.1__

SIGNATURE __JS__ DATE __8/30/86__

VERIFICATION: (CORRECTION TESTED)

CLOSURE TEST NO. __RE-FS-37__

SIGNATURE __DC__ DATE __8/30/86__

Figure 2

XYZ, INC.

REQUEST FOR CHANGE
F-1001 12/77

RFC RE-86-34

Name Jane Doe	Date Prepared 9/5/86	Page 1 of 1	Date 9/5/86
Position Handicapped elevator rider	Telephone No. x7611	Receiving DLC	
Department Name and Location Software QA, Bldg. 55		DLC Location	

Module/Unit/System Affected
RUN_ELEVATOR

☐ HARDWARE	☒ DESIGN	☐ DEVIATION	☐ EMERGENCY
☒ SOFTWARE	☐ COMPATIBILITY	☐ COST REDUCTION	☒ URGENT
☐ FIRMWARE	☐ MFG. OPTION	☐ OTHER	☐ ROUTINE

I use a wheelchair, and have trouble pressing the 'Door Open' button on the computer-controlled elevator and then getting out without being squashed by the doors.

Make the door stay open longer when the button is pushed.

	Signature	Title	Date
☒ ACCEPTED FOR INVESTIGATION	Q___	CHAIR, CCP	9/6/86

☒ APPROVED ☐ REJECTED	UM/SDM	Date
Action Taken or Reason for Rejection 'Door Open' delay extended from 1 second to 3 seconds.	ECM/SCM	Date
	CPPM	Date

Figure 3

XYZ, INC. INFORMATION AND ASSISTANCE REQUEST
F-7092 1091

	INCIDENT NUMBER
	One

IAR NO. 61623

	CENTRAL SERVICE NAME AND LOCATION		DATE	CUSTOMER TEL. NO. & EXT.	CUSTOMER NO.
TO	RTH		2/5/87	999-555-5555	H-345

	CUSTOMER/COMPANY NAME AND ADDRESS	ORIGINATOR NAME ☐ XYZ ☒ CUSTOMER
FROM	Death Rays, Inc.	Sam Short

ORIGINATOR TEL. NO. & EXT.	SALES DISTRICT NO
999-555-5551	

SE ZONE NO.	SYSTEM DIV	OPERATING SYSTEM	RELEASE & PATCH LEVEL	SOFTWARE CODE	FIRMWARE RELEASE AND PATCH LEVEL	DEVICE TYPE/MODEL NO

MEMORY SIZE	TYPE DISC	APPLICATION NAME	RELEASE & PATCH LEVEL	SOFTWARE CODE	PROGRAM NAME	MODIFIED ☐ YES ☒ NO	IF YES EXPLAIN BELOW
		RUN_ELEVATOR	2.0				

DOCUMENTATION ENCLOSED

RUN_ELEVATOR Installation Manual

_____

_____

_____

_____

PROBLEM DESCRIPTION

When the elevator call button on the 14th floor is pushed, the elevator tries to continue upward when it arrives, refusing to open the door and running the motor.

THIS SECTION BELOW SHOULD BE COMPLETED ONLY BY XYZ CENTRAL SERVICES

DATE AND TIME RECEIVED	ASSIGNED TO	PRIORITY CODE
2/7/87	John Smith	Highest

RESPONSE CODE	DATE MORE DOCUMENTATION REQUESTED	DATE DOCUMENTATION RECEIVED	DATE OF RESPONSE	SPAR DATE	SPAR NO.	SGU CODE
			2/7/86			

PROBLEM RESPONSE

The software was designed and coded to operate in a 14 story building. Death Rays, Inc. occupies a building where the floors are numbered 1,2,...12,14. There is no floor numbered 13, therefore there are only 13 floors. The hardware interrupt requesting floor 14 causes the elevator to rise, counting as it goes, so it tries to find a 14th floor where none really exists.

Next time let's send someone over to look at the building before we sign off the requirements!

_____

SYSTEMS ENGINEER'S SIGNATURE

Figure 4

Examples of Software Change Forms

James E. Tomayko
The Wichita State University

The following five pages are examples of forms used in industry for customers and developers to report software problems and request changes. Included are:

1. Discrepancy Report Form
2. Request for Change Form (1)
3. Request for Change Form (2)
4. Software Problem Report Form (Boeing)
5. Information and Assistance Request Form *(used by customers only)*

Pages 11 through 15 are unnumbered.

DISCREPANCY REPORT

Fill in bold areas. Please print.

Report No.:

ORIGINATOR	Name:	Date Prepared:	Date Needed:
	Position:	Mail Address:	Version:
	Group:	Module/Unit/System Affected:	

TYPE OF PROBLEM

☐ Design ☐ Deviation

☐ Compatibility ☐ Other

PRIORITY

☐ Emergency
☐ Urgent
☐ Routine

DESCRIPTION

ANALYSIS

CORRECTION

DOCUMENTATION

Description of changes:

RESOURCES (EST.)

Programmer Hours: _____
Computer Expenses: _____
Other: _____
TOTAL COST: _____

☐ ACCEPTED FOR INVESTIGATION   Signature   Title   Date

FINAL DECISION

☐ APPROVED   ☐ REJECTED
Action Taken or Reason for Rejection

QA: Date

Monitor: Date

V&V: Date

Implementor: Date

REQUEST FOR CHANGE

Fill in bold areas. Please print.

Request No.:

ORIGINATOR

Name:	Date Prepared:	Date Needed:
Position:	Mail Address:	Version:
Group:	Module/Unit/System Affected:	

TYPE OF CHANGE

☐ Design ☐ Deviation
☐ Compatibility ☐ Cost Reduction
☐ New Feature ☐ Other

PRIORITY

☐ Emergency
☐ Urgent
☐ Routine

DESCRIPTION OF PROBLEM

PROPOSED SOLUTION

☐ ACCEPTED FOR INVESTIGATION Signature Title Date

FINAL DECISION

☐ APPROVED ☐ REJECTED

Action Taken or Reason for Rejection

DOCUMENTATION

Description of changes:

RESOURCES (EST.)

Programmer Hours: _____

Computer Expenses: _____

Other: _____

TOTAL COST: _____

QA: Date

Monitor: Date

V&V: Date

Implementor: Date

XYZ, INC.

REQUEST FOR CHANGE
F-1001 12/77

_____ RFC _____

ORIGINATOR

Name	Date Prepared	Page 1 of _____	Date
Position	Telephone No.	Receiving DLC	
Department Name and Location		DLC Location	

Module/Unit/System Affected

TYPE OF CHANGE

☐ HARDWARE	☐ DESIGN	☐ DEVIATION	☐ EMERGENCY
☐ SOFTWARE	☐ COMPATIBILITY	☐ COST REDUCTION	☐ URGENT
☐ FIRMWARE	☐ MFG. OPTION	☐ OTHER	☐ ROUTINE

DESCRIPTION OF PROBLEM

PROPOSED SOLUTION

Signature	Title	Date

☐ ACCEPTED FOR INVESTIGATION

FINAL DISPOSITION

☐ APPROVED   ☐ REJECTED

Action Taken or Reason for Rejection

UM/SDM	Date
ECM/SCM	Date
CPPM	Date

☐ ERROR

☐ IMPROVEMENT

BOEING
SOFTWARE PROBLEM REPORT

SPR NO: _____

REF NO: _____
(UER-PRR-ECP)

PROBLEM: ORIGINATOR'S NAME _____

SYSTEM
FAILING _____

VERSION
NO. _____

FILE/MODULE
NAME _____

RELATED
SYSTEMS _____

RELATED
TEST CASE _____

CLASSIFICATION: PROBLEM DESCRIPTION: _____

O SOFTWARE
O HARDWARE
O COMMENT/POL
O DOCUMENTATION
O INFORMATION ONLY

CORRECTION
REQUIRED BY:
ANALYSIS TO BE
DONE BY ORG _____ GROUP _____ SIGNATURE _____ DATE _____

ANALYSIS: (PREPARED BY RESPONSIBLE SOFTWARE DESIGN ORGANIZATION) RECEIVED DATE _____

EXPLANATION: _____

CLASSIFICATION:

O CODING
O DESIGN
O COMMENT/POL
O DOCUMENTATION
O ENVIRONMENT
O NO S/W CHANGE REQD
O REPORTED ON _____
  PREVIOUSLY
O OTHER CONFIG
  AFFECTED
  (SPR NO. _____)

SIGNATURES: _____ ANALYST _____ DATE _____ ORIGINATOR _____ DATE _____

CORRECTION: (BRIEF DESCRIPTION OF WORK AND LIST OF MODULES CHANGED)

SYSTEM WORK _____

SIGNATURE _____ DATE _____

DOCUMENTATION WORK: (BRIEF DESCRIPTION OF DOCUMENTATION WORK) _____

S/W D NO. _____ PAGE NO. _____ DCN/ADCN NO. _____ DATE _____
S/W D NO. _____ PAGE NO. _____ DCN/ADCN NO. _____ DATE _____
S/W D NO. _____ PAGE NO. _____ DCN/ADCN NO. _____ DATE _____
S/W D NO. _____ PAGE NO. _____ DCN/ADCN NO. _____ DATE _____

CLOSURE APPROVAL SIGNATURES:

_____ DATE _____

CONFIRMATION: (CHANGE VERIFIED BY S.C.M.)	VERIFICATION: (CORRECTION TESTED)
MTM NO _____	CLOSURE TEST NO. _____
AVAILABLE IN VERSION _____	
SIGNATURE _____ DATE _____	SIGNATURE _____ DATE _____

XYZ, INC. INFORMATION AND ASSISTANCE REQUEST

F-7092 1081

INCIDENT NUMBER

IAR NO. 61623

TO	CENTRAL SERVICE NAME AND LOCATION	DATE	CUSTOMER TEL. NO. & EXT.	CUSTOMER NO.

FROM — CUSTOMER/COMPANY NAME AND ADDRESS

ORIGINATOR NAME ☐ XYZ ☐ CUSTOMER

ORIGINATOR TEL. NO. & EXT.	SALES DISTRICT NO.

SE ZONE NO.	SYSTEM DIV.	O	OPERATING SYSTEM	RELEASE & PATCH LEVEL	SOFTWARE CODE	FIRMWARE RELEASE AND PATCH LEVEL	DEVICE TYPE/MODEL NO.

MEMORY SIZE	TYPE DISC	A	APPLICATION NAME	RELEASE & PATCH LEVEL	SOFTWARE CODE	PROGRAM NAME	MODIFIED ☐ YES ☐ NO IF YES EXPLAIN BELOW

DOCUMENTATION ENCLOSED

PROBLEM DESCRIPTION

_____

_____

_____

_____

_____

THIS SECTION BELOW SHOULD BE COMPLETED ONLY BY XYZ CENTRAL SERVICES

DATE AND TIME RECEIVED	ASSIGNED TO	PRIORITY CODE

RESPONSE CODE	DATE MORE DOCUMENTATION REQUESTED	DATE DOCUMENTATION RECEIVED	DATE OF RESPONSE	SPAR DATE	SPAR NO.	SGU CODE

PROBLEM RESPONSE

_____
SYSTEMS ENGINEER'S SIGNATURE

1 - ANALYST COPY

Example of a
Software Configuration Management Plan

Walter Smith, Jon Lange
Carnegie-Mellon University

The authors of this example configuration management plan were undergraduate students in the course Software Engineering, taught by James E. Tomayko at Carnegie-Mellon University, fall semester, 1986. This plan was submitted as part of a class project to develop an operations simulator for a manned Mars research station. Two implementations were specified: one written in Ada, the other in Pascal, in widely differing development environments.

Pages 17 through 29 are unnumbered.

Mars Research Station OpSim
Configuration Management Plan

Walter Smith

Jon Lange

CMP-1 DRAFT of 23 September 1986

Table of Contents

1. Introduction

This plan is based on ANSI/IEEE Standard 828-1983, section 3. Also used were "Outline for a Configuration Management Plan for Computer Programs" and "Software Requirements, Baselining, and Control", both from Data and Configuration Management Workshops of the Electronics Industries Association.

There are two teams for this project, each with an identical Configuration Management organization. Both teams will use this plan. In some areas, the teams will differ; for example, they will use different revision control procedures. These differences will be noted below.

1.1 Purpose

The purpose of this plan is to describe and define the policies and proceedings to be used in the application of configuration management to the development of the Mars Research Station Operational Simulator (Mars OpSim).

1.2 Scope

This plan provides for the application of configuration identification, control, and status accounting during the development of Mars OpSim. Included is the assignment of item numbers to the Computer Program Configuration Items (CPCI), revision control during development, procedures to be followed to ensure interface control, and identification of status accounting techniques and procedures.

1.3 Definitions and acronyms

The terms relating to configuration management used in this document are defined by EIA Configuration Management Bulletin No. 4A, *Configuration Management for Digital Computer Programs.*

2. Management

2.1 Organization

The Configuration Manager will report directly to the contract monitor. Responsibility and authority has been delegated to the Configuration Manager by the contract monitor to act for him in all matters relating to the implementation of Configuration Management in accordance with this plan.

The Configuration Control Board (CCB) is appointed by the Configuration Manager for the purpose of evaluating all proposed changes to released specifications, computer programs, manuals, design documents, listings, and other items identified for formal change control.

2.2 CM responsibilities

The Configuration Manager is responsible for

- Writing the Configuration Management Plan.

- Appointing the Configuration Control Board and presiding over its meetings.

- Creating the necessary forms for CM procedures.

- Maintaining easily-accessible central repositories for the current versions of the software configuration items in machine-readable form.

- Maintaining records of CM activity.

- Preparing releases of configuration items and providing for their distribution.

- Receiving change requests and discrepancy reports and presenting them to the CCB.

- Ensuring that approved changes are made and recorded.

- Conducting audits of the CM process.

2.3 SCMP implementation

2.3.1 Configuration Control Board

The Configuration Control Board will consist of five members: the Configuration Manager (chairman), the Quality Assurance Manager, the Project Administrator, and one member each of the Design and Coding teams. One member will be selected by the board to be secretary, and will keep minutes of each meeting.

2.3.2 Central repositories

There will be central repositories maintained by the Configuration Managers for the current releases of all software configuration items.

2.3.2.1 Documentation repositories

Two directories will be maintained on TOPS machines for copies of the current releases of documentation items. Current releases of all Ada documentation will be kept in the <WSON> directory on TOPS-D (td.cc.cmu.edu). Current releases of all Pascal documentation will be kept in the <JL13> directory on TOPS-C (tc.cc.cmu.edu). Copies of general project documentation will be kept in both directories. Filenames will use the naming conventions of Section .

2.3.2.2 Code repositories

The Unix RCS system will be used to manage the Ada team's code. A central RCS directory on the project VAX will be maintained for source code version control. In addition, a directory containing copies of the current release of source code, object code, and libraries will be maintained.

The current releases of Pascal source code, object code, and libraries will be kept in the <JL13> directory on TOPS-C. Backup floppies will be kept by the Configuration Manager.

2.3.3 Releases

A release of a configuration item will result in the updating of that item in the appropriate repository, the notification of all project members, and possibly the distribution of human-readable copies.

Releases will be made periodically; the period between releases will depend on the frequency and urgency of changes made. A release will be made immediately prior to each review.

2.3.4 Change requests and discrepancy reports

Change requests and discrepancy reports will be given to the Configuration Manager for introduction at the Configuration Control Board meetings. A central mailbox will be established for this purpose.

2.4 Applicable policies, directives, and procedures

2.4.1 Release policy

Software will be released and placed under formal change control at the formal request of the group responsible for developing the software; for example, a Coding team member might send electronic mail to the CM at the completion of a module's unit testing. After software has been released, it will not be changed unless as a result of an approved change request. Periodically, as determined by the CCB, a new release containing the changes will be made, and copies will be placed in the release directory.

2.4.2 RCS usage policy

RCS will be used to maintain all released Ada source code. Those groups working on Ada software are strongly encouraged to use RCS internally as well.

The RCS Header string will be used in all Ada source files in a manner that ensures the inclusion of the header string in the object code. The RCS Log feature will be used in all Ada source files within comments.

2.4.3 Pascal coding policy

It will be the responsibility of those working on Pascal software to use the most recent available versions of released software, and to keep their working versions up to date.

2.4.4 Standard routine headers

Standard templates to be provided by the Configuration Managers will be used for the opening comment of all modules and routines to ensure consistency in the information provided. The actual templates will be designed in cooperation with the Design and Coding teams and made available in the release directories, they may not necessarily look like the samples

below.

2.4.4.1 Sample Ada file header

```
-----------------------------------------------------------------
-- Mars OpSim
--
-- Module:       Queue
-- File:         queue-insdel.a
--
-- Routines:
--      Queue.Insert     Insert an element into a queue
--      Queue.Delete     Delete an element from a queue
--
-- $Header: queue-insdel.a,v 1.1 86/10/23 04:23:33 wrs $
-- $Log:         queue-insdel.a,v $
-- Revision 1.1  86/10/23  04:23:33  wrs
-- SCR 25 ; Fixed MemFull exception bug in Queue.Insert
--
-- Revision 1.0  86/10/11  01:29:22  wrs
-- Initial release
--
-----------------------------------------------------------------
```

2.4.4.2 Sample Ada routine header

```
---------------------------------------------------------------
--
-- Routine:     Queue.Insert
-- Author:      Joe Programmer
-- Function:    Insert an element into a queue.
--
-- Inputs:      q        The queue
--              elt      The element to be inserted
-- Outputs:     q        (modified)
-- Exceptions:  MemFull
-- Globals used: none
--
-- Specification:
-- This routine inserts elt at the tail of q.  If there is
-- insufficient memory, the MemFull exception is raised.
--
-- Implementation:
-- Memory is allocated for elt.  If there is insufficient...etc.
--
-- Side Effects:
-- none
--
-- Modification history:
-- 1.1  86/10/23  wrs
--       SCR 25 ; Fixed MemFull exception bug.
--       Missed post-allocation memory compaction problems.
-- 1.0  86/10/11  wrs
--       Initial release
--
---------------------------------------------------------------
```

2.4.4.3 Sample Pascal file header

```
{ Mars OpSim

Module:      Queue
File:        queue-insdel.pas

Routines:
        Queue_Insert    Insert an element into a queue
        Queue_Delete    Delete an element from a queue

Last Edit:   10/23/86 by jll

Revision History:
1.1  10/23/86  jll
        SCR 25 ; Fixed MemFull error bug in Queue_Insert
1.0  10/11/86  jll
        Initial release
}
```

2.4.4.4 Sample Pascal routine header

```
{
Routine:        Queue_Insert
Author:         Joe Programmer
Function:       Insert an element into a queue.

Inputs:         q       The queue
                elt     The element to be inserted
Outputs:        q       (modified)
Globals used:   none

Specification:
This routine inserts elt at the tail of q.  If there is
insufficient memory, the MemFull handler is called.

Implementation:
Memory is allocated for elt.  If there is insufficient...etc.

Side Effects:
none

Modification history:
1.1  86/10/23  jll
        SCR 25 ; Fixed MemFull error bug.
        Missed post-allocation memory compaction problems.
1.0  86/10/11  jll
        Initial release
}
```

3. SCM Activities

3.1 Configuration identification

3.1.1 Naming conventions

3.1.1.1 Documentation

Documentation configuration items will be identified by a two-part code D-R where D is a code for the document and R is the release number of that document. Releases will be numbered consecutively from one. Unreleased versions of documents will have an additional number identifying the last change applied. This number will start at one and be incremented by one at each change. For example, the second release of the Software Requirements Document would be called SRD-2; after five changes had been applied, it would be called SRD-2-5.

Filenames for the machine-readable versions of documents will follow the same conventions; for example, the Scribe source for SRD-2-5 would be called SRD-2-5.MSS. On Unix systems, lower-case letters will be used: srd-2-5.mss.

3.1.1.2 Code

Source code modules will be identified by a module name not longer than ten characters. Releases of source code modules will be numbered consecutively from one, and will be called "*ModuleName* Release n", where n is the release number. Unreleased versions of source code modules will have an additional number identifying the last change applied. This number will start at one and be incremented by one at each change. For example, the second release of the *Queue* module would be called "Queue Release 2"; after five changes had been applied, it would be called "Queue Release 2.5". RCS version numbers of source files will be made to correspond to module release numbers.

3.1.2 Configuration items

The configuration items for this project (and their codes, where applicable) will be

- Software Requirements Document (SRD)
- Software Specifications Document (SSD)
- Preliminary Design Document (PDD)
- Software Design Document (SDD)
- Software Test Plan (STP)
- User Document (UD)
- Individual software modules (as defined by the SSD)
- Individual software source files (as defined by the SDD)
- Individual object code files (as defined by the SDD)

3.1.3 Baselines

Six baselines will be defined: Requirements, functional, allocated, design, product, and operational. Releases will update these baselines.

3.1.3.1 Requirements baseline

The requirements baseline is established at the Requirements Review. It consists of the Software Requirements Document.

Once the requirements baseline is complete, the general specifications are created, establishing the functional baseline.

3.1.3.2 Functional baseline

The functional baseline is established at the Specifications Review. It consists of the Software Requirements Document and the Software Specifications Document.

Once the functional baseline is complete, the preliminary design specifications are created, establishing the allocated baseline.

3.1.3.3 Allocated baseline

The allocated baseline is established at the Preliminary Design Review. It consists of the Preliminary Design Document, the Software Test Plan, the User Document, and the components of the functional baseline.

This baseline allows the detailed design process to begin. Once all software components have been designed, the design baseline is established.

3.1.3.4 Design baseline

The design baseline is established at the Critical Design Review. It consists of the Software Design Document and the components of the allocated baseline.

This baseline starts the actual process of coding and debugging. As each routine completes its unit-level test, it will be released, establishing the product baseline.

3.1.3.5 Product baseline

The product baseline is established by the integration of all software components and the release of the software to the Test and Evaluation group. It consists of all of the configuration items.

Once all design descriptions have been validated against the requirements and all software components have passed acceptance tests, the operational baseline is established.

3.1.3.6 Operational baseline

The operational baseline is established when the product has passed acceptance testing and has been released to the customer. It consists of all of the configuration items.

3.2 Configuration control

Configuration control is the systematic evaluation, coordination, and approval or disapproval of proposed changes to a baseline. Formal control of the configuration of an item begins with the definition and release of a baseline for that item.

3.2.1 Change classification

Class I	A Class I change is defined as any change which affects a customer approved requirements, functional, allocated, product, or operational baseline.
Class II	A Class II change is defined as any change which is not Class I or any change which corrects errors in

the documentation of a customer-approved baseline.

All Class I changes must be approved by the contract monitor. Class II changes can be approved by the Configuration Control Board. The contract monitor may override the CCB's classification of any change.

3.2.2 Configuration Control Board

The CCB will review all change requests and discrepancy reports. The total impact of each request will be evaluated by the board, taking into consideration appropriateness, cost, technical feasability, scheduling constraints, effects on other items, and effects on testing.

Class II changes will be approved or disapproved by the CCB. Class I changes for which the CCB recommends approval will be forwarded to the contract monitor for final approval or disapproval. The CCB will designate an implementor for each fully approved change.

The CCB will be convened on a weekly basis or as deemed necessary by the Configuration Manager. Copies of change requests and deficiency reports will be made available to CCB members prior to meetings for examination and evaluation. Minutes of each meeting will be distributed to all project members.

A joint meeting of the Ada and Pascal CCB's may occasionally be called to discuss matters of mutual concern.

3.2.3 Change Control Documentation

Two documents will be used to process and control changes: the Software Change Request and the Software Discrepancy Report. The Change Request will be used for requesting a change to a released configuration item that is an improvement rather than a repair. The Discrepancy Report will be used for requesting a change to a released configuration item necessary because of a failure of the item to meet requirements.

3.2.4 Change processing

A change request will be processed in the following manner:

1. A change request is prepared.

2. It is transmitted to the Configuration Manager, who numbers the change for tracking purposes and notifies the Configuration Control Board.

3. The Board classifies the change and may then

   - approve a Class II change, or approve the submission of a Class I change to the contract monitor

   - disapprove the change, in which case the change dies

   - modify the proposed change, in which case the modifications are made and the change is reevaluated

4. If a change is approved, it is implemented by someone designated by the Board.

3.2.5 Discrepancy Report processing

A discrepancy report will be processed in the following manner:

1. A discrepancy report is prepared.

2. It is transmitted to the Configuration Manager, who numbers it for tracking purposes and notifies the Configuration Control Board.

3 The Board may immediately reject the report, in which case it dies, or assign someone to analyze the report and prepare a correction.

4. If the report is not rejected, the board may approve, disapprove, or modify the proposed correction.

5. If the correction is approved, it is implemented by someone designated by the Board.

3.3 Configuration status accounting

The following status accounting logs and reports will be maintained:

Configuration Item Index
> The Configuration Item Index will list each configuration item along with its creation date, current released version, and the versions of its component items.

RCS change logs   RCS change logs will be maintained for each Ada source file. They will show the history of the changes made to the source files, as well as their release histories.

Pascal change logs Each pascal source file will contain a change log showing the history of the changes made to it, as well as its release history.

Discrepancy reports
> All Software Discrepancy Reports will be retained.

Change requests   All Software Change Requests will be retained.

3.4 Audits and reviews

The Configuration Manager will periodically audit the system to ensure that policies and procedures are being followed. Audits will be conducted as follows:

Requirements   At the Requirements Review, the CM will release the Software Requirements Document and place it under change control.

Functional   At the Specification Review, the CM will release the Software Specifications Document and place it under change control.

Allocated   At the Preliminary Design Review, the CM will review the Preliminary Design Document to assign configuration items according to the defined software components, and update the Configuration Item Index. The CM will also release the Preliminary Design Document, the Software Test Plan, and the User Document, and place them under change control.

Design   At the Critical Design Review, the CM will review the Software Design Document to assign configuration items according to the defined routines, and update the Configuration Item Index. The CM will also release the Software Design Document and place it under change control.

Product   The CM will review the software to ensure that all routines are fully updated and have been tested and released to the Test and Evaluation group.

Operational   The CM will ensure that all software components are fully updated and have passed the appropriate acceptance tests.

4. Records collection and retention

All records as defined in Section 3.3 will be available for inspection on request. Written logs, change requests, and discrepancy reports will be maintained in a binder by the Configuration Manager. RCS logs and Pascal revision logs will be available as part of the source code.

All written memoranda to and from the Configuration Manager will be maintained in the same binder. Electronic mail will be retained in a log file.

Typical Revision Control System Session

James E. Tomayko
The Wichita State University

The following annotated typescript is a typical session with the Revision Control System (RCS) tool that runs on Unix. It demonstrates the tasks needed for checking in and out controlled modules, shows how the simultaneous update problem may be prevented, and shows how version trees may be created.

```
Script started on Thu Aug  7 10:20:46 1986

 1 /usr0/jet/adaprograms] emacs synch.a
```

The editor is commanded to open an Ada source file synch.a, *which is about to be placed under configuration control. The configuration manager enters the following at the top of the file:*

```
        --$Header$
        --$Log$
```

This action marks for RCS the location in the file to place the initial header and later version logs.

```
 /usr0/jet/adaprograms] ci synch.a
```

This command tells RCS to check in the module. RCS prompts for a description of the configuration item as follows:

```
synch.a,v  <--  synch.a
initial revision: 1.1
enter description, terminated with ^D or '.':
NOTE: This is NOT the log message!

>> This is a task which synchronizes two simultaneously executing tasks.
>> Programmer: James E. Tomayko
>> .

sh: /bin/snoop: not found
done


 3 /usr0/jet/adaprograms] ls
ada.lib          adaone.a.BAK      calculator.a     synch.a,v        synch.lib
adaone.a         adaone.a.CKP      realtime
```

Note in this display of the file names that the module synch.a *is gone, replaced by* synch.a,v, *which is where the original file and the deltas of the revised versions of the file will be kept.*

```
 4 /usr0/jet/adaprograms] co -l synch.a
```

This command checks out synch.a *for revision. The* -l *option locks the file and prevents any other program-*
mer from checking out the same module. RCS replies as follows:

```
synch.a,v  -->  synch.a
revision 1.1 (locked)
sh: /bin/snoop: not found
done
```

Note in this listing of the files that now there is a synch.a *checked out of* synch.a,v:

```
5 /usr0/jet/adaprograms] ls
ada.lib         adaone.a.CKP    synch.a         synch.lib
adaone.a        calculator.a    synch.a,v       typescript
adaone.a.BAK    realtime
```

When the file is opened for editing, the Header and Log comments are found to have been modified:

```
6 /usr0/jet/adaprograms] emacs synch.a
```

```
--$Header: synch.a,v 1.1 86/08/07 10:21:40 jet Exp $
--$Log: synch.a,v $
```

```
Revision 1.1  86/08/07  10:21:40  jet
Initial revision
```

After the changes have been made to the file, it is checked in, at which time RCS prompts for a description of
the changes:

```
7 /usr0/jet/adaprograms] ci synch.a
synch.a,v  <--  synch.a
new revision: 1.2; previous revision: 1.1
enter log message:
(terminate with ^D or single '.')
```

```
>> Added to comments to incomplete exception handling statements.
>> .
```

```
sh: /usr/local/lib/rdiff: not found
sh: /bin/snoop: not found
done
```

The next time the module is checked out, the change comments appear in the Log:

```
8 /usr0/jet/adaprograms] co -l synch.a
```

```
synch.a,v  -->  synch.a
revision 1.2 (locked)
sh: /bin/snoop: not found
done
```

```
9 /usr0/jet/adaprograms] emacs synch.a
```

```
--$Header: synch.a,v 1.2 86/08/07 10:24:37 jet Exp $
--$Log: synch.a,v $

Revision 1.2  86/08/07  10:24:37  jet
Added to comments t. incomplete exception handling statements.

Revision 1.1  86/08/07  10:21:40  jet
Initial revision
```

The changes made to the module during this check out actually created a new line of development, so instead of checking the module in as Version 1.3, RCS is directed to check it in as Version 2.0:

```
10 /usr0/jet/adaprograms] ci -r2.0 synch.a

synch.a,v  <--  synch.a
new revision: 2.0; previous revision: 1.2
enter log message:
(terminate with ^D or single '.')

>> Created a divergent form of synch.a by adding a third process to be
>> synchronized.
>> .

sh: /usr/local/lib/rdiff: not found
sh: /bin/snoop: not found
done
```

Future modifications to Version 1.2 can be made, thus keeping alive its branch of the version tree, by specifying it in the co command using the −r option. Actually, any version of the module can be recovered.

More information about RCS can be found in [Tichy82].

Presentation Support Viewgraphs

James E. Tomayko
The Wichita State University

The following viewgraphs have been used by the author to support teaching software configuration management in an industrial short course setting. They appear in the order of use.

1. The Role of Configuration Management
2. Functions of Configuration Management
3. Commitment to Configuration Management
4. Typical Configuration Items
5. Configuration Management Library Functions
6. Types of Change I
7. Types of Change II
8. Fundamental Principles to Guide Configuration Control Boards
9. Factors Determining Configuration Control Board Characteristics
10. Hierarchies of Configuration Control Boards
11. Key Factors in Evaluating Change $_1$
12. Key Factors in Evaluating Change $_2$
13. Key Factors in Evaluating Change $_3$
14. Discrepancy Report Evaluation Process Flowchart
15. Change Request Evaluation Process Flowchart
16. Simultaneous Update Problem
17. Version Tree
18. Trend Analysis
19. Standards For Configuration Management Plans

Pages 34 through 52 are unnumbered.

The Role of Configuration Management

Development Disciplines

- Specification
- Design
- Code
- Test

Management Disciplines

Product Integrity

Controlling Disciplines

- Software Quality Assurance
- Software Configuration Management
- Independent Verification and Validation
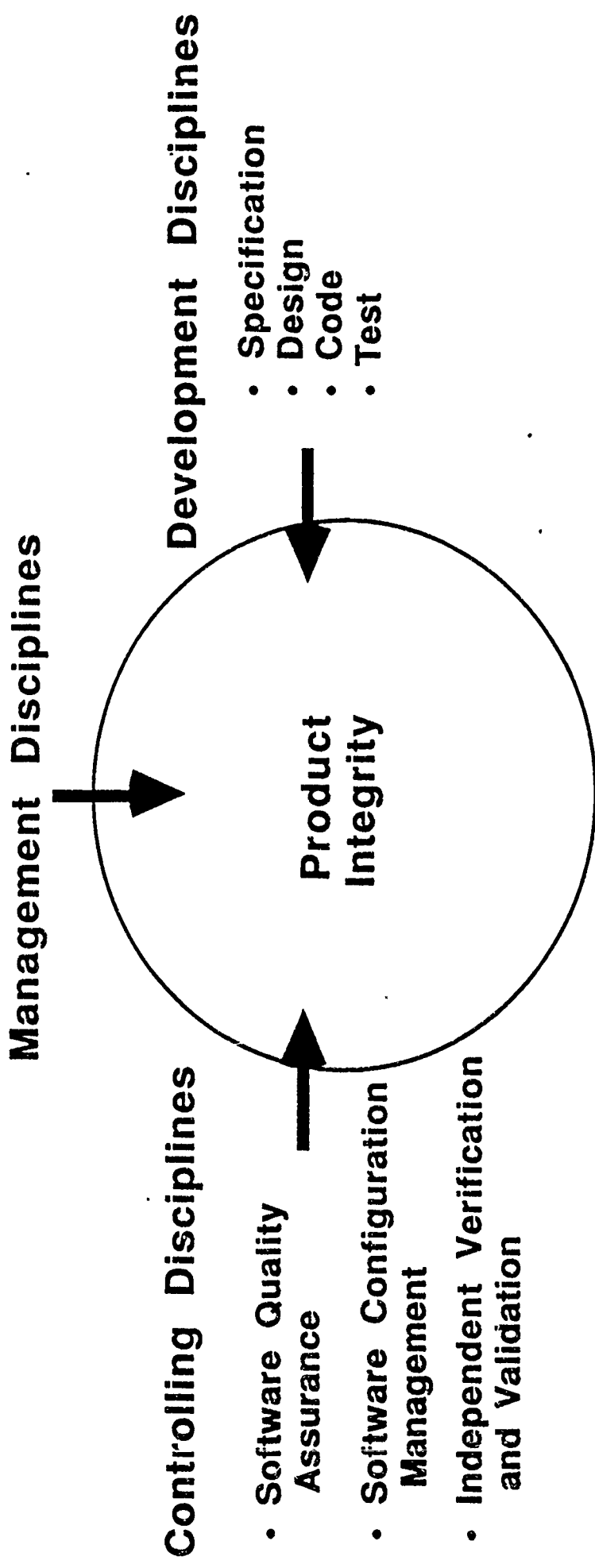
Adapted from Bersoff, Henderson and Siegel, <u>Software Configuration Management.</u>

Functions of Configuration Management

- Maintain Integrity of Configuration Items

- Evaluate and Control Changes

- Make the Product Visible

Commitment To Configuration Management By The Entire Organization Is The Key To Its Success.

Typical Configuration Items

- Requirements

- Specifications

- Design Documents

- Source Code

- Object Code

- Test Plans

- Test Suites

- User Manuals

- Maintenance Manuals

- Interface Control Documents

- Memory Maps

Configuration Management Library Functions

- Software Part Naming

- Configuration Item Maintenance/Archiving

- Version Control

- Preparation of Product for Release

Types of Change I:

- Discrepancies

  - Requirements Errors

  - Development Errors

  - Violations of Standards

Key Factors in Evaluating Change[1]

- Size

- Complexity

- Date Needed

- CPU and Memory Impact

- Cost

- Test Requirements

Key Factors in Evaluating Change2

- Criticality of the Area Involved

- Politics (Customer/Marketing Desires)

- Approved Changes Already in Progress

- Resources (Skills, Hardware, System)

- Impact on Current and Subsequent Work

Factors Determining Configuration Control Board Characteristics

- Hierarchies

- Scope

- Composition

Hierarchies of Configuration Control Boards

Product: B-7777 Transport Plane
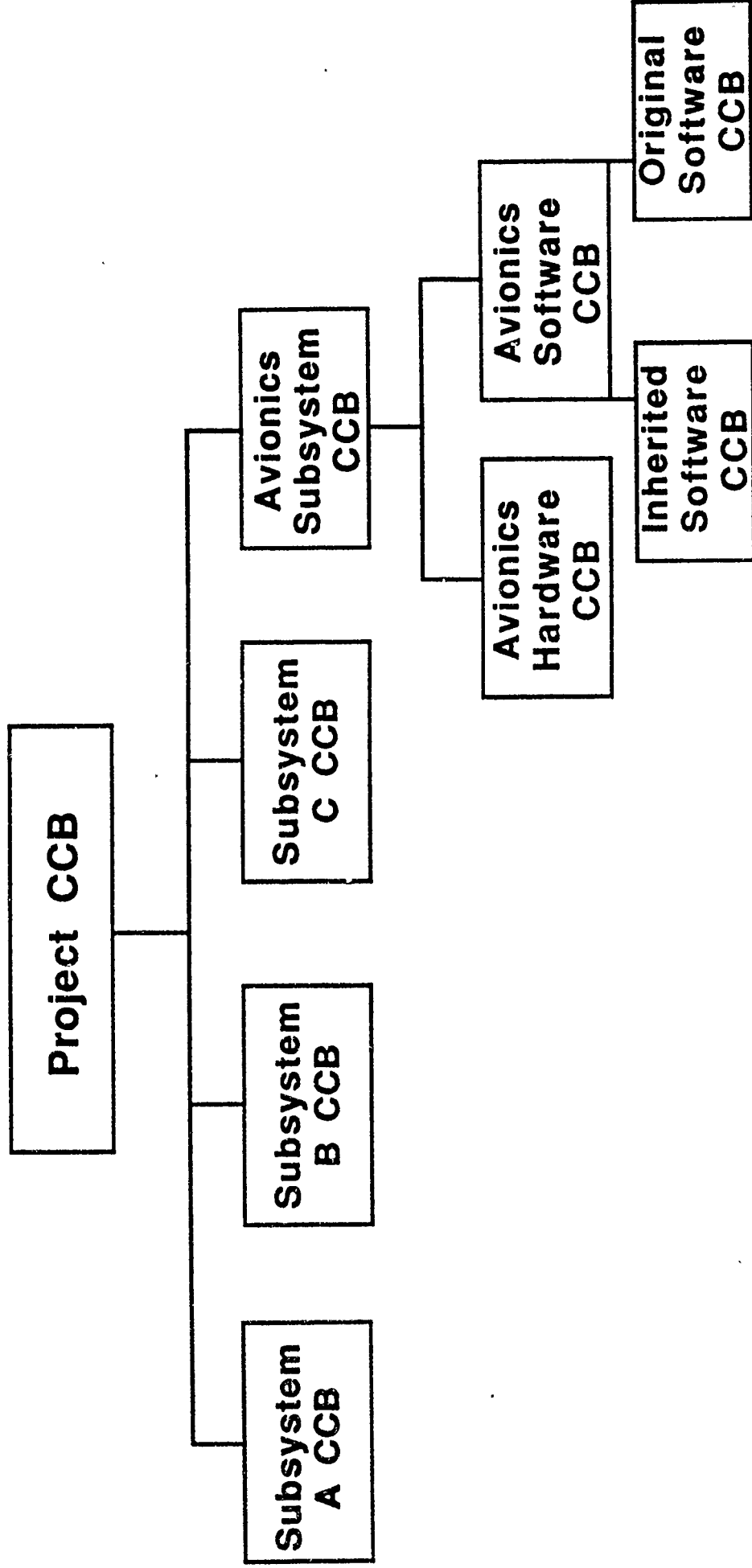
Key Factors in Evaluating Change3

- *Is There An Alternative?*

Discrepancy Report Evaluation Process Flowchart

Change Request Evaluation Process Flowchart

```
CR Submitted  →  CR Logged  →  CCB Evaluation
```

CCB Evaluation — No → CR Closure Audited & Logged

CCB Evaluation — Authorized → Development Group Makes Change → Configuration Items Updated → CR Closure Audited & Logged
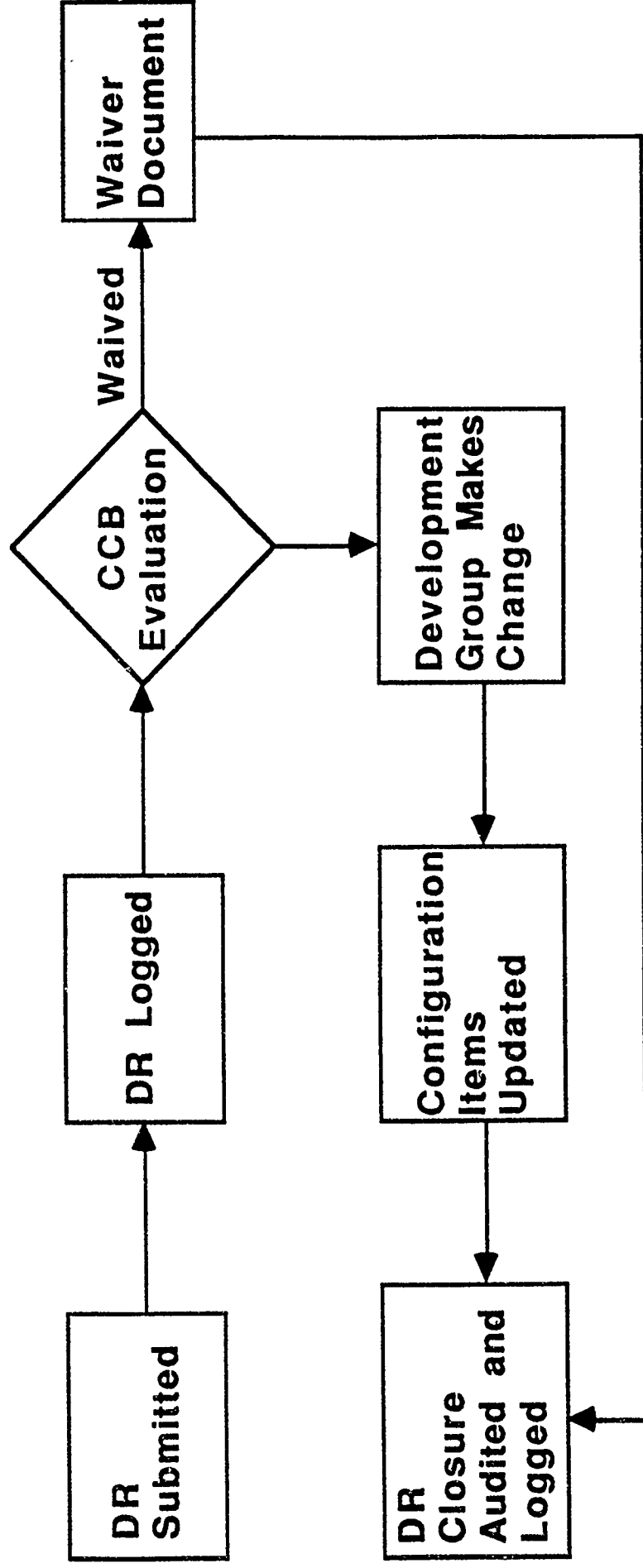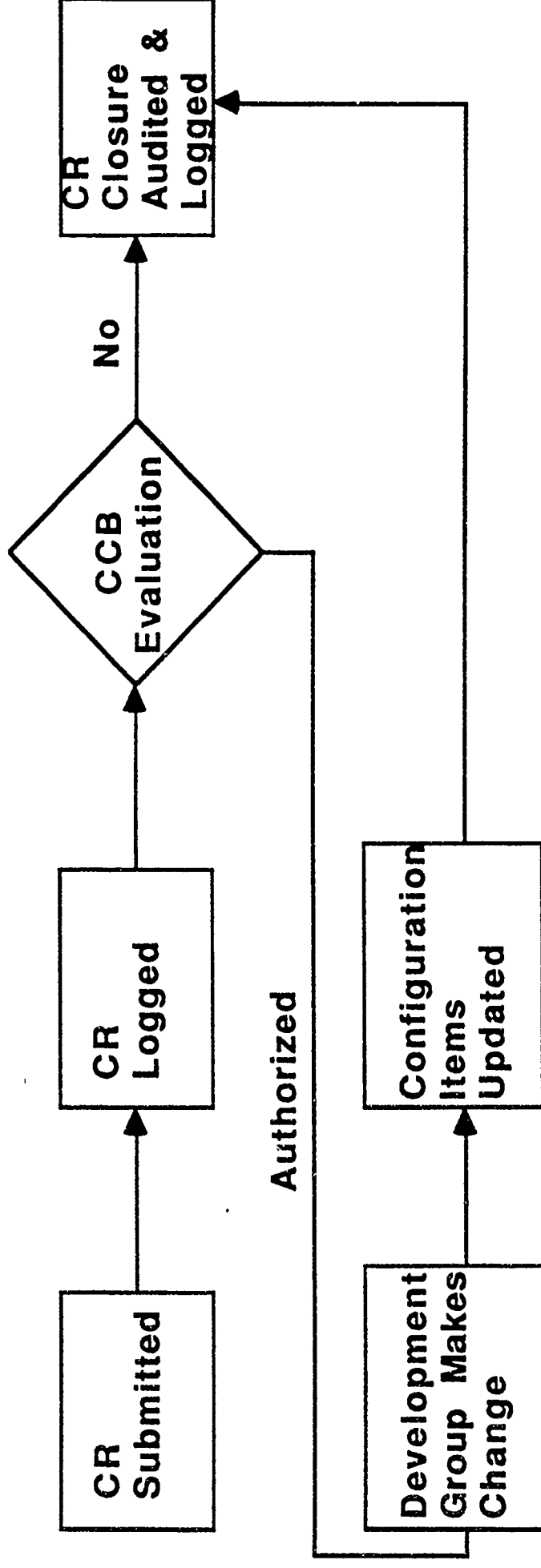
Simultaneous Update Problem
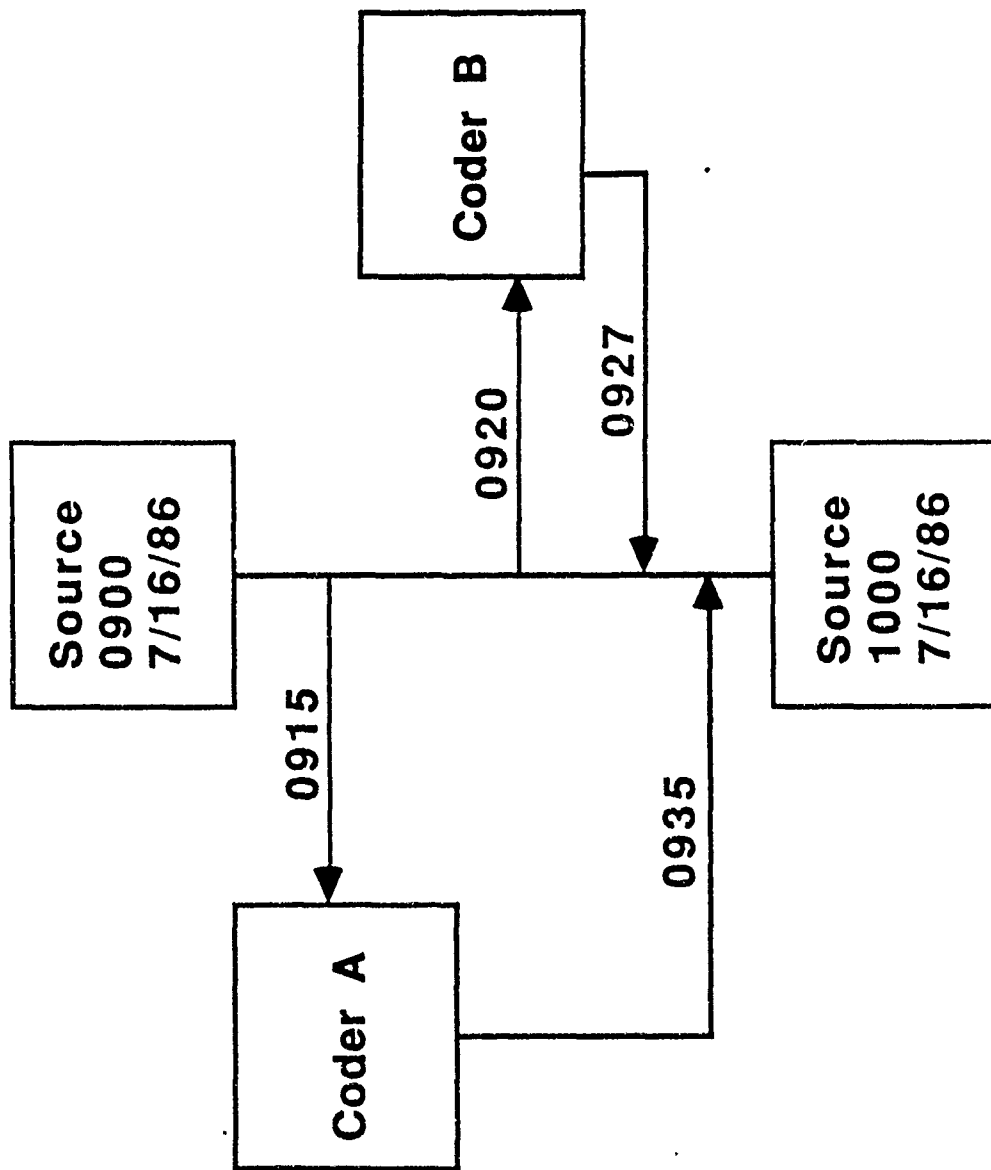
Version Tree

Trend Analysis

RFC

DR

1986

400

300

200

100
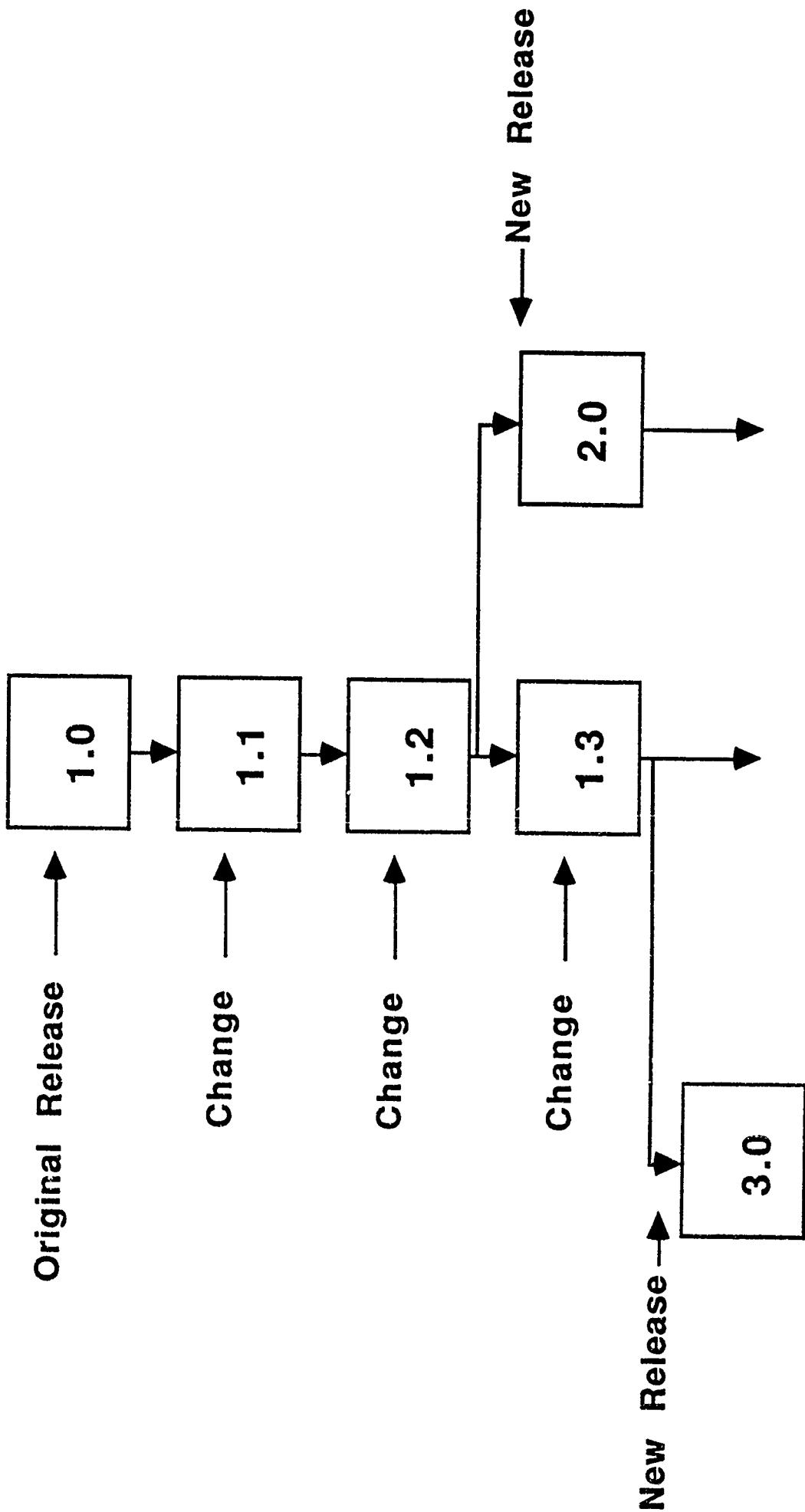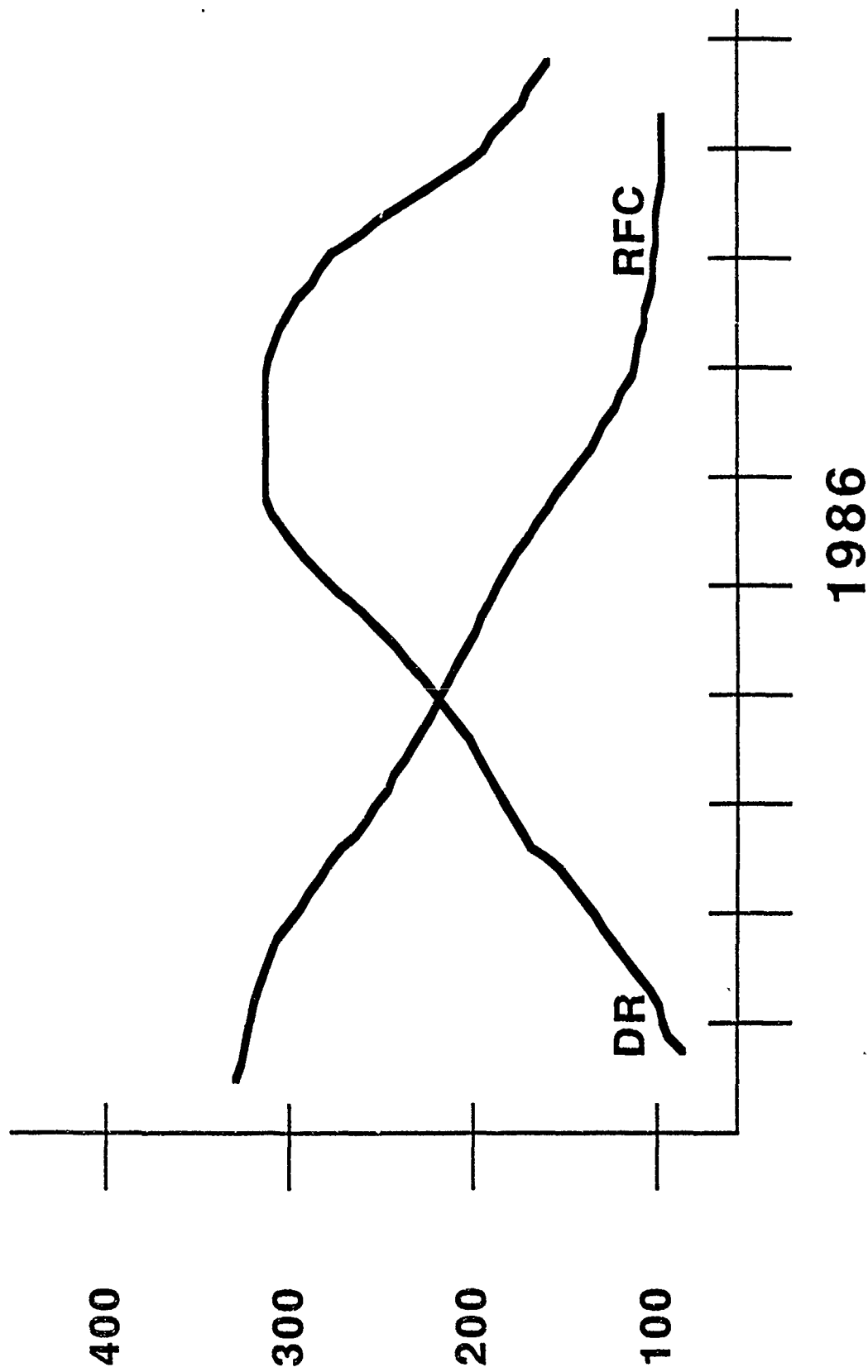
Standards For Configuration Management Plans

- IEEE [ANSI/IEEE 828-1983]

- DoD
  - MIL-STD-483A (Air Force)
  - DoD-STD-2167

Sample Examinations

James E. Tomayko
The Wichita State University

The following exams have been developed to test achievement of the behavioral objectives in the Software Configuration Management curriculum module SEI-CM-4-1.0 (Preliminary).

Exam 1

1. List two ways that software changes throughout the life cycle:

2. What is a configuration item?

3. How does configuration control maintain the integrity of configuration items?

4. Identify the configuration items of a typical product.

5. Define the term 'baseline.'

6. Give an example of a non-ambiguous software part numbering/naming scheme.

7. What is the difference between discrepancies and requested changes?

8. What is the difference between discrepancies caused by requirements errors and those caused by development errors?

9. List three key items included in a discrepancy reporting form.

10. List three key items included in a change request form.

11. Show how discrepancy reports and change requests are tracked within a software development organization.

12. List and define the fundamental principles of implementing change control boards.

13. Define the scope of change control boards of at least three levels of a product development organization.

14. A change control board is given responsibility over the avionics subsystem of a digitally-controlled aircraft. Who should be on the board? Who should make the final decisions?

15. List five considerations specific to evaluating the repair of discrepancies.

16. List five considerations specific to evaluating change requests.

17. List five considerations specific to evaluating requests for new derivatives of a product.

18. Specify how the implementation of changes can be tracked.

19. Define the simultaneous update problem.

20. Define the concept of version trees.

21. Identify at least three necessary characteristics of good version control tools.

22. List at least three commercially available version control tools, their similarities and differences, and their suppliers.

23. Identify at least two standards for configuration management plans.

24. List three items in an effective configuration management plan.

25. List at least three personal characteristics needed by effective configuration management personnel.

Exam 2

1. Responding to approved change requests and _____ are two ways that software changes throughout the life cycle.

2. Documents and code placed under configuration control are referred to as _____.

3. List three items under configuration control in the course of a typical software development:_____.

4. When an item is placed under configuration control is often referred to as a _____.

5. Discrepancies are:_____.

6. Requested Changes are:_____.

7. Discrepancy reporting forms should have line items such as:_____.

8. Change requests and discrepancy reports are tracked within an organization by:_____.

9. What should be considered when evaluating the repair of discrepancies?

10. List three considerations in evaluating change requests that differ in importance from the list in #9.

11. Define the simultaneous update problem.

12. Why are different versions of a product often needed?

13. What characteristics should a good version control tool have?

14. Where can guides to developing configuration management plans be located?

15. List three items in an effective configuration management plan.

16. What are System Description Languages?

Summary of the SEI Workshop on
Software Configuration Management

Katherine E. Harvey
Software Engineering Institute

Participants

Bradley Brown	Boeing Military Airplane Co.
James Collofello	Arizona State University
Robert Glass	Seattle University
Ted Keller	IBM Federal Systems Division
Richard Parten	Lockheed
Mary Shaw	SEI
Howard W. Tindall	Martin-Marietta
James E. Tomayko	SEI (host)

Introduction

The following is a summary of the discussion during the Software Configuration Management meeting held at the Software Engineering Institute in Pittsburgh on July 16, 1986. In this document I have tried to determine the major concerns brought up and the conclusions reached during the day long discussion. The discussion ran in many directions, often changing topics quickly and not returning to the original subject for quite some time. Therefore, I did not try to summarize the discussion chronologically, since I felt that would be more confusing than informative. I have, instead, tried to sort the various concerns and conclusions into specific areas and summarized the discussion of each major point brought up in those areas.

Overview

Definitions of Software Configuration Management. The basic definitions of Configuration Management that the workshop participants more or less agreed upon seems the best place to start; since the definition is fundamental to the discussion of Software Configuration Management (SCM). Jim Tomayko, the host of the workshop, put as his capsule description of Configuration Management: "the disciplines and techniques of initiating, evaluating, and controlling change of software products during and after the development process." This definition met with general approval, although the discussion as a whole brought out a much more complex and discerning description. The most apparent concept missing from the original definition was that SCM is a fundamental and *essential* management tool for software development projects. It is more a management concept than a concrete structure and is invaluable to the organized and rapid output of a software product.

Although the concept of SCM was felt to be fundamental to the maintenance of software products, the workshop members felt that associating SCM with maintenance is misleading. Configuration management should not start simply when a software product reaches the maintenance phase; the whole development process must be managed in such a way that SCM can work properly. For instance, if the original designer does not document his work properly, then the configuration management process breaks down; because later changes create problems not immediately apparent based on the existing documentation. SCM therefore is an integral part of the entire software design and development process and a vital part of all software engineering.

The state of Software Configuration Management today. One of the major points of discussion was how Software Configuration Management was being utilized by the software engineering community today. No one at the meeting felt that SCM was being effectively utilized as a management tool; in fact, just the opposite. Although there have been many corporations with solid SCM programs, there are a vast number of companies producing software today with either no program whatsoever or a programs that hinder rather than help. What is wrong with the SCM programs today?

A major problem is the lack of a *widespread* under-

standing of the usefullness of a solid SCM program. Although man, large companies do have configuration management systems, often when they turn a project over to a smaller contractor, the Software Configuration Management is left up to that contractor, who often chooses to do nothing. If the configuration management is bad, one can almost guarantee that the documentation will be bad. Then, when the development process for a software product is over and it goes into maintenance mode, the contractor turns over a software project with incomplete documentation. So the company left with the project is lost, they start playing around with it, and they are left with "spaghetti" software. According to the members of the workshop, this kind of thing happens all the time; even though many of these projects are expected to interact with others.

One key factor in an effective configuration management system is a solid Configuration Control Board structure. However, in most companies today the importance of the boards and their members is overlooked. Often the people put into these boards do not have the training or experience to make decisions about changes or problems in software products. One example brought up, was an entire Software Configuration Management division that exists but is virtually a hindrance to the organization. In this organization when a change request is written, often only a paragraph or less of information, and sent into headquarters, it goes to the Configuration Control Division. However, this division's job is simply to put a number on the CR and send it out; without any kind of board meeting or discussion whatsoever. This CR is then sent out to people who can't possibly tell from a couple of sentences of information whether or not the change is a good idea. Then, after several weeks, the request is sent back with "nonconcur" or "concur" stamped on it, and oftentimes it takes months before any real action is taken on the document. If the change is approved it goes to the implementation organization that writes the functional specification and the detail design with no review of either document. This same organization does all of the coding and testing, without ever consulting a review board or the originator of the request; then when i⊦ finally shows up in the field the originator proba ` won't even recognize it.

Many times the people at higher levels of large projects don't understand software and think of it as simply "another subsystem". It becomes a difficult task to convince these project managers that on a large complex system, or *any* project that has as its root a data system, the software is an integral function of the project. Often in these projects the use-

fulness of a configuration management system is overlooked and therfore this vital management tool is not used properly. Many times attempts are made to use other methods, like the CSSR or C Spec. system, to maintain control over projects. But these programs tend to be cursory measurements of progress and costs without ever getting down to the real work of change management. It is in the configuration boards that changes are discussed and information gets moved around; where sleeves are rolled up and the nuts and bolts of the software are laid out.

So it is very apparent that there is a great need for improvement in the area of Software Configuration Management. . The problems are large and widespread; of course they won't be solved overnight. However, the workshop participants had a great many ideas about the components of an ideal configuration management system. These may provide the base for educating future software engineers to better manage their projects through Software Configuration Management.

The Software Configuration Management System

A general picture. The workshop more or less agreed that there are too many unpredictable circumstances in the corporate world to build a generic all purpose configuration management structure. However, it is possible to sketch in certain key elements without creating a definitive structure. Some of the elements are just fundamental characteristics of a SCM system, while others are more subtle details that will create a more efficient management machine.

All large system projects have systems of Change/Configuration Control Boards (hereafter known as CCB's). The structure of the CCB's may differ widely according to the type of project, the company in charge, and the size or cost of the program. However, the CCB's are a necessary element to almost any SCM structure. These boards work in a kind of waterfall structure with a great number of control boards at the lower levels which feed up into the higher levels. Change usually bubbles up from the bottom where the programming activity is boiling. At times there is reverse traffic when change requests come down from either the customer or the system manager, but the vast majority comes from the area of the most programming activity. Therefore the greatest number of

boards is at this bottom level of great activity, and these boards should be the most active.

Ideal CCB characteristics. One of the most important characteristics for any control board, but especially the lower level boards, is that they should be *active*. Because this is where information is passed around, where you begin to see the project's shape and direction, it is vital that the boards be a well-used and functioning body in the SCM structure. The CCB should be a place of discussion, where *any* problems or requests that come up in a project get hammered out. On large projects these board meetings often last longer than a full day, but the work being done in them is vital.

Because this work is so vital to a project, "casual involvement" simply cannot exist in the CCB system. It is important that the management people on each board look into every CR/DR that comes before them. Even if the change or bug is presented in a very casual or non-mission critical way, it is the duty of the board members to look into it as if it were. If board members allow the casual nature of a presentation affect their decisions and evaluations, problems may be overlooked that could escalate later into emergency situations.

There should be a route for emergency changes so that the system won't break down during emergency situations. There should also be a CCB appeal route. This means that it would be possible to go to a higher CCB if the originator of the change request deemed it to be absolutely necessary to reverse the original boards decision. This will help to keep the board meetings from becoming "shouting matches", and help people discuss things rationally. However, the appeal route must be carefully controlled, (perhaps by upper level boards making decisions as to which appeal request should be accepted) in order to keep the authority of the lower boards intact.

It is important not to limit the number of boards because of past SCM practices or "efficiency". It is actually more efficient to have as many boards as possible within cost and common sense parameters. Each board should have the minimum number of people possible needed to make decisions. Therefore each CCB's jurisdiction should be well documented, and only those people directly involved in or affected by changes in their jurisdiction need to be at their CCB meetings. This way only vital people are involved in their particular CCB decisions, and other important people who not directly involved don't waste their time.

Ensuring proper documentation. At a very basic level Configuration Control should be involved with

all changes taking place at the project level using a lot of discussion and review for *each* change being made. In order for this to happen, documentation throughout the development phase of a system must be enforced. So in every SCM structure it would be a good idea to have a division that will make sure that the original developers are writing down *everything*. Documentation never gets done by those developing the code without outside influence and almost never gets done post-facto (certainly not accurately). If there is no documentation, there is nothing to control. So the documentation "enforcers" are a good idea for a strong SCM system, provided their authority is well documented and strictly monitored.

These various characteristics of a good SCM structure may vary a great deal; especially when the existing authority levels are very different. The authority hierarchy in a company or program has a great deal to do with the configuration management system, and all the elements that have been talked about so far rest upon well-organized authority levels.

Authority in SCM Systems

Authority hierarchies. Although, as previously stated, the CCB's are places for discussion it must be stressed at this point that the final decision-making authority should lie with one individual. The control boards do not vote on changes; one person makes a decision under advisement. This is extremely important when trying to avoid interproject politics and keep a program oriented toward its proper goal. The higher level boards have greater authority, of course, than the lower boards and the system level CCB belongs at the top of the pyramid. It is the head of the system level CCB who has authority to make the final judgements on CR/DR's and any last minute emergency "patches", although this authority is usually delegated to lower level boards who are more often confronted with the problems as they come up. This means that whoever is making those final decisions had better be pretty sharp or the program is headed for trouble.

It is also healthy to a project to have a slight adversary relationship existing between the software design manager and the head of the program. The design manager will be fighting for what he needs for his particular area, while the head of the program should be seeing a more overall picture, hardware systems as well as software. If both these people are

well trained in project management, then the adversary relationship will provide much needed checks and balances within the project system.

The various CCB's should have documentation readily available to them detailing what specific areas over which they have authority. Each board needs to be sure what decisions they have authority over and how much authority they have to make a decision. When CR/DR's come up, there should never be confusion as to who is responsible for looking into them. So it is very important that CCB jurisdiction and authority cover *every* area at some level, especially those critical to the project, and this authority *must* be documented. For example, if the Testing and Evaluation division discovers a DR, it must be clear whether they have the authority to make changes in the program or they need the authority of a higher board to make the change; and whether this authority changes in the event of a mission critical DR. At the workshop two experiences were given as examples: in one situation the Test/Evaluation people did have authority to make changes even on non-mission critical DR's, while in the other situation they simply reported the DR's to higher boards for action or the Evaluation people simply figured out ways to work around non-mission critical DR's. The responsibility for these decisions need to be well documented to avoid confusion.

Key authority concepts. It is very important to understand that authority levels cannot be generically structured to fit any situation. Usually the structure of any given SCM system depends on the authority levels already in existence in the particular company or program involved. Any project manager coming into a company or program must have a good appreciation for the existing authority hierarchy. The Software Configuration Management system that he is going to instigate, reorganize, or make additions to, must be molded around those authority levels.

Oftentimes the way people perceive problems can create difficulties. While one person may see something as a "problem", someone else may see it as simply a "change". Who has the authority to deal with these varying perceptions? It may be that it comes under the authority of each CCB's head, or that an entirely different division or CCB should be set up to deal with this question. Once again, this will probably depend on the authority hierarchy already existing. However, it may also depend on the people in the program, the size of the project, and various other management considerations.

It is also important for each company that goes under contract with another to have appreciation for the existing authority hierarchy in the other company or organization. When everyone involved in a project understands the authority structure and the way they are expected to work within it, a smoother operation and a more productive work atmosphere sill result.

Tools for Software Configuration Management

Tools of the trade. Quite a few methods for maintaining control over change were discussed. Many were technical devices that are well documented and available, so the group spent very little time on these. Others were not discussed necessarily as "tools", but I felt that they could be labeled as a specific tool for software configuration management and that this would be a good place to summarize them. First, let's look at the naming and/or numbering of products.

It was felt that one of the most important aspects of any system for naming software products was that it be specific as well as *not* ambivilent. A specific example was brought up regarding NASA back when the name of a software system matched the mission for which they were being built. This, however, soon became a problem. In these projects there is usually a very long time between when one starts building a software system and the time the mission it is intended for finally flies. Often halfway through the maintenance life-cycle of this software, major changes are made in the project; payloads may be swapped or scrapped, as may the mission vehicles, and so on. When these changes are made, the name of the mission is often changed. Then, one is left with a software system named for a mission that may not fly for years if it ever goes up at all. It is easy to see how this could become confusing. Therefore, the software is now named and numbered in a completely separate way so that there can be no relevance to the mission for which its being developed. What is important to see in this example is that the naming system had to be adjusted to become more specific to the product as well as less ambivilent.

Because various divisions may have different names for a single system, and because communication must eventually extend beyond the purely technical community, it is important to be able to see how the nomenclature evolved and how the various nomenclature relate to one another. IBM uses a waterfall diagram to show the path of each particular system and its various names along the way. But

perhaps more importantly, IBM puts on the same page a cross reference list. Since each nomenclature may for a particular software build have three different names with each of these names understood by different divisions, the cross reference list is important for clear understanding and communication. Using diagrams is also a useful tool for communicating with those outside the technical community.

A management tool that might not be distinctly thought of as a SCM "tool", is that of using "freeze dates" when putting out incremental releases of a software system. The example at the meeting went something like this: Usually, the top management people on a project are very anxious to see some sort of working software even though the software designers are still working out the bugs in a code and may be very reluctant to release it. In a case like this, having freeze dates for the software to be turned in will force the developers to release what they have even if they feel it is incomplete. Usually the first release will be chaotic but this will give a good idea about where to go and what needs to be fixed, and the consumer has a working product. Even if it has a lot of DR's, having a completed product is a positive incentive and will improve the working atmosphere. The freeze dates must be rigid; if the developers don't get their projects in on time, they won't be included in the release. If it isn't enforced the computer people will keep fiddling around and changing things, and the entire program will fall behind schedule. Once again it is important to remember that implementing a tool like this will depend a *great* deal on the existing situation.

Documentation and Credibility

Documentation. At one point in the day's conference Jim Tomayko asked the group if they knew whether *anyone* paid attention to the standards for software configuration management put out by IEEE. No one at the meeting had even heard of them. They were aware of the Department of Defense Standard 2167, but it was generally acknowledged that this was overlooked by most program managers. The standards get overlooked because the rigorous documentation requirements that they establish are seen as cumbersome and so documentation does not get enforced. At first, this seems easier for both the managers and developers. It isn't until they are waste deep in the mire of unmet schedules, undocumented software with hundreds of unseen DR's, rising costs, and consumers anxious to

see this project that is now out of control, that the importance of enforced documentation standards is apparent. How do you motivate people to use cumbersome standards when they haven't been "burned" by past experience? Obviously, standards for software configuration management are a useful tool, but getting people to use them is another matter.

It cannot be said enough that without documentation there is nothing to put under configuration control. There *must* be a valid functional specification document in order to get past the Preliminary Design Review or there is nothing to put under the management system and you're already off schedule. A brief list of documentation includes:

- requirement specification documents,
- functional specification documents,
- detail design documents,
- user manuals,
- maintenance manuals,
- interface control documents,
- memory layouts,
- test plans,
- and the code itself, of course.

All of this, plus more not mentioned here, comes under maintenance control, unless it is subject to a *project specific* waiver. Because many of these documents are scrapped when a product reaches maintenance phase, it would be useful perhaps to maintain a configuration index for each product so that enough documentation is maintained for configuration control during the maintenance phase.

Even in the essential area of documentation there must be consideration for the project involved. If the project is large the managers are usually more careful about enforcing documentation because the project as a whole is probably being approached in a very careful and cautious way. However, in a smaller project SCM tends to take a back seat and the documentation, therefore , doesn't seem important or economical. Sometimes full rigor on the SCM and documentation can be relaxed slightly on a smaller project, but then you need someone in command who knows when full rigor can be relaxed and when it should be enforced. However, good documentation will always help configuration management people to make sounder judgements and more credible evaluations.

Credibility.. Good basic documentation is the basis for Configuration Control Board evaluations on the issues before them. In order for CCB's to make in-

telligent, rational, and credible decisions on CR/DR's there is certain data that is necessary. This data should be well documented so that the CCB evaluation of the data will carry weight. This list of necessary data was developed at the workshop:

- The size of the change.
- Are there alternatives? Would it be relatively simple to work around whatever is being changed?
- The complexity of the change. Does it reference other systems? Does this system support other software or rely on other support software that would need to be changed accordingly?
- The need date. Basically, the board needs to know how much time they would have to make the change and test it, before the consumer needs a working project.
- Impact. This is related to its complexity. What kind of effect will this change have on subsequent work. The board needs to look down the road a bit and see where the project is going.
- Cost. How much will the change cost? Also, will this change save money in the overall project?
- The criticality of the area. NO CR/DR's can be overlooked if the problem will prove to be mission critical. All other areas of evaluation should be rethought if the bug might possibly create critical problems.
- Other CR's under current evaluation. Will another change solve this problem or do other more critical changes rely on this software remaining the same.
- Test requirements. This area takes in how much testing will be needed which will affect the costs and time needed for the change.
- Resources. Do you have the people available to work on this program? Do you have the hardware equipment available to use for this change?
- CPU and memory impact.
- Benefit. How much of an advantage will it be to change the software?
- Politics. In the corporate and commercial world, it would be good idea to evaluate *who* is asking for the change and whether or not the board's decision might be used as a bargaining point in the future.

- Maturity of the change. How long has the change been before the board? If it is still considered to be worthwhile to change something after a long time has passed, then the board should consider it more carefully.

By using this data, you can often minimize the number of "side effects" that the changes you are making will have. Even if the side effects are unavoidable, the use of this carefully documented evaluation process may help to identify where those effects are going to be. Of course, at this time it is impossible to be absolutely sure that all the side effects have been discovered. For example, suppose there are two changes that are being made at the last minute in an emergency situation and they are each tested and evaluated. It is possible that although they may have no real side effects on the system *separately*, when they are "patched" in at the last minute they may have serious side effects *together*. This is the greatest fear when dealing with late patches, but careful documentation and evaluation of the data involved in each change may help to alleviate some of the guesswork.

In the purely commercial arena, credibility is the key in dealing with the marketing division or the customer. If they have a change request that is going to create more difficulty than it is worth, the configuration management people should be able to show *documented data* that will make the control board's evaluation credible. When a customer can see the kind of impact a change is going to have on the time, size, or cost involved in a project, they will better understand and more readily accept the management's decisions regarding the project. The key is a thorough and well documented evaluation based on the previous listed data. The list can be changed or expanded on, according to the needs of the project, but as it stands, it gives a fairly accurate picture of the kind of information that is going to be needed for credible evaluation.

SCM and the Real World

Going from the classroom to the corporation. Two points came up early in the meeting that helped to categorize many of the problems discussed later in the day.

1. We live in an irrational world, but computer science and software engineering is based on concrete and rational logic. How does one make this rational knowledge fit into an ir-

rational world? Software Engineering and Design is not just like it is in the textbooks.

2. Very often the existing system dictates what kind of changes take place and what kind of configuration management is used, rather than the ideal or proper software design practices.

Both of these concepts are difficult to teach to young, inexperienced software engineers who are coming directly from the classroom. They are concepts that are usually learned through experience. A software engineering graduate expects to put the principles of Software Configuration Management directly into effect. Suddenly they are confronted with an irrational world that does not easily follow the logical course of configuration management. It isn't the mainline textbook problems that are going to throw an educated software engineer; instead, it's the small peripheral problems that build up and take control of a project. These little things, the result of this irrational world, include corporate politics, unforeseeable accidents, human personalities, and day to day unexpected emergencies. Also, a new program manager may have to deal with a system that does not follow regulation SCM practices and does not want to change. Often corporations have become comfortable with a particular structure that does not have room for SCM, and it can be quite frustrating to a young manager to be asked to comply with "company policy" rather than smart software configuration management.

There are some attributes of the irrational world and some system protocol specifications that will never be able to be changed, regardless of a software engineer's chagrin when dealing with them. Learning to deal with these inexplicable and usually frustrating areas of SCM requires experience in the world where they exist. A textbook will never be able to adequately transfer the kind of knowledge needed to deal with the irrational world, There will always be people who will be able to manage corporate software configuration better than others, regardless of classroom performance---another result of that irrational world.

A few well educated configuration management personnel are not going to make much of an impact on Software Configuration Management today. There must be a way to communicate the concepts of SCM to a great number of people involved in the development of software products, even the people who are not directly responsible for the configuration management of a particular system (this would include everyone from the computer scientist writing

the actual code to the final consumer of the product). If a few concepts of SCM are known by a majority of people dealing with the development of a software product, then people will be able to function more smoothly within the system and the whole process will be tighter. This is also important when you remember the large number of companies that are using contracts and subcontracts with other companies. Unless the concepts of SCM are widespread among many companies, Software Configuration Management will be dependent upon whether a subcontractor chooses to use SCM or not.

Two Perspectives. One last point that seemed to be emphasized at the meeting was that of two perspectives emerging. It has been previously stated that when a company goes under contract with another to develop a software system, the management people should have respect for the existing SCM structure in the contracting company. The two perspectives are (1) that of the originator of the project and (2) that of the contractor that goes into this program. NASA is a good example. They will often put several companies under contract for a single mission and these companies often turn around and subcontract another company to work on various parts of the system. NASA has a very structured system for configuration management, and the companies under contract often have SCM systems of their own. It is very clear to see that being able to see the "give and take" needed in a situation like this. Each company needs to try and comply with the SCM demands of the contract originator. When a software engineer is trying to take the concepts of SCM into the real world, he should be prepared to deal with these perspectives.

Conclusions

It is apparent to me that Software Configuration Management courses are essential to progress within Software Engineering today. SCM is tied to every stage of software product development. A good configuration management team could make the difference between products coming in on time, within cost and coming in late, full of bugs, with greater costs. Education seems to be the place to *start*, but there seems to be much more involved than classroom development alone. It seemed that what the group was trying to do was begin a program that would teach software engineers that they *need* to learn the concepts of Software Configuration Management wherever that education amy be avail-

able (whether learning in the classroom or gaining experience in the field). I would conclude that what seems to be wrong in Software Configuration Management today is that too many software engineers don't seem to think they are missing much without a solid knowledge of SCM. If they can be shown the importance of SCM, then perhaps they will be more eager to learn its concepts and to use it more often and more effectively in the software development field today.

Bibliography on
Version Control and Configuration Management

Daniel Conde

Digital Equipment Corporation

This is a bibliography of documents related to the problem of version control and configuration management. Specifically, it concentrates on the problem of System Modelling, which is loosely defined to be the task of:

> Giving programmers help in describing the structure of large systems: getting consistent versions of files, replacing single modules within a running system, and recompiling and rebinding just what has been changed, all in the right order. (*From a Xerox internal glossary*)

Most of the documents do not attack the whole problem just defined, but together they represent work done on many aspects of the problem. Some are on various programming languages that help the construction of large systems. Others refer to specific systems designed to help programmers in the problem of version control, or configuration management. Some even try to solve many of these problems in a coherent way. This problem has gained attention recently as the problems of larger projects written by many programmers are realized. Some of the recent efforts were reported in the proceedings of the ACM Software Engineering Symposium on Practical Software Development Environments and the GTE Workshop on Programming-in-the-Large that are listed here. Some, but not all of the papers from those proceedings are referenced here. This year's ACM Software Engineering Symposium should promise to present more recent work. The list is by no means complete. I have mainly included documents that are publically (*sic*) available. I have also not included internal memos or any works in progress. Since I expect many new works this year, this serves to capture some of the early work. I will try to augment it as in the future. This bibliography is derived from a version I once distributed on-line, but I have deleted all reverences to various internal memos and added new references.

I hope this reading list will help those planning to build systems to solve this problem, or those who want to apply ideas into their existing environment. It is not required to run a program to help in system modelling, as various methods and conventions are sometimes sufficient. A program does help programmers automate the process.

[ACM84]
Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.. Pittsburgh, PA, 1984.

[Allman81]
Allman, Eric. *An Introduction to the Source Code Control System, Project Ingres.* UC Berkeley, 1981.

[Avakian82]
Avakian, Arra. The Design of an Integrated Support Software System. *Proceedings of the SIGPLAN 82 Symposium on Compiler Construction.* ACM, June, 1982, 308.

[Belady76]
Belady, L.A. and M.M. Lehman. A Model for Large Program Development. *IBM Syst. J. 15*, 3 (1976), 225.

[Belady78]
Belady, L.A. *Large Software Systems.* IBM Thomas J. Watson Research Center, Jan., 1978.

[Belady79]
Belady, L.A. The Characteristics of Large Systems. In *Research Directions in Software Engineering,*

P. Wegner, ed.MIT Press, 1979, 106-142.

[BellLabs81]
Source Code Control Systems User's Guide, in UNIX System III Programmer's Manual. AT&T Bell Laboratories, AT&T Information Systems, 1981.

[Bianchi76]
Bianchi, M.H., and J.L. Wood. A User's Viewpoint on the Programmer's Workbench. *Proceedings of the 2nd Conference on Software Engineering.* ACM and IEEE, 1976, 193.

[Cooprider79]
Cooprider, Lee. *The Representation of Families of Software Systems.* CMU-CS-79-116, Carnegie-Mellon Univ., April, 1979.

[Cristofor80]
Cristofor, E., T. Wendt, and B. Wonsiewicz. Source Control + Tools = Stable Systems. *Proceedings of Compsac 80.* IEEE, Oct., 1980.

[Dahl68]
Dahl, Ole-Johan. *Simula 67 Common Base Language.* Norwegian Computing Center, Oslo, 1968.

[DEC82]
CMS/MMS: Code/Module Management System Manual. Digital Equipment Corporation, 1982.

[DeRemer76]
DeRemer, Frank, and H. Kron. Programming-in-the-Large versus Programming-in-the-Small. *IEEE Trans. Software Eng. 2,* 2 (June 1976), 80-86.

[Deutsch80]
Deutsch, L. Peter, and Ed Taft. *Requirements for an Experimental Programming Environment.* CSL-80-10, Xerox PARC, June, 1980.

[DOD80]
Department of Defense. *Stoneman: Requirements for Ada Programming Support Environments (APSE).* DoD, Feb., 1980.

[Erickson83]
Erickson, V.B. Build A Software Construction Tool. *AT&T Bell Laboratories Technical Journal 63,* 6 (Aug. 1983).

[Estublier84]
Estublier, J. Preliminary experience with a Configuration Control System. *Proceedings of SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development.* ACM, May, 1984, 149.

[Estublier85]
Estublier, J. A Configuation Manager: The Adele Data Base of Programs. *Proceedings of the Workshop on Software Environments for Programming-in-the-Large.* GTE Laboratories, June, 1985, 140.

[Feldman79]
Feldman, Stuart I. Make - A Program for Maintaining Computer Programs. *Software Practice and Experience 9,* 4 (April 1979), 255-265.

[Fritzon85]
Fritzon, Peter. The Architecture of an Incremental Programming Environment and some Notions of Consistency. *Proc. Workshop on Software Engineering Environments for Programming-in-the-Large.* GTE Laboratories, June, 1985, 64.

[Gandalf85]
J. Sys. and Software 5, 2 (May 1985). Issue dedicated to the Gandalf System..

[Glasser78]
Glasser, Alan L. The Evolution of a Source Code Control System. *Software Engineering Notes 3,* 5 (Nov. 1978), 122.

[Goldstein80a]
Goldstein, Ira, and Danny Bobrow. Representing Design Alternatives. *Proc. Artificial Intelligence and Simulation of Behaviour Conference.* Amsterdam, July, 1980.

[Goldstein80b]
Proc. First Annual Conference of the National Association of Artificial Intelligence.. Stanford, California, Aug., 1980.

[Goldstein80c]
Goldstein, Ira, and Danny Bobrow. *A Layered Approach to Software Design.* CSL-80-5, Xerox PARC, Dec., 1980.

[GTE85]
Proc. Workshop on Software Engineering Environments for Programming-in-the-Large.. Harwichport, Massachusetts, June, 1985.

[Habermann76]
Habermann, Nico A., Lawrence Flon, and Lee W. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Comm. ACM 19, 5* (May 1976), 266.

[Habermann79a]
Habermann, Nico. *A Software Development Control System.* Carnegie-Mellon Univ., 1979.

[Habermann79b]
Habermann, Nico. Tools for Software Construction. *Proc. Software Tools Workshop.* Boulder, Colorado, May, 1979.

[Habermann80]
Habermann, Nico. *System Decompositions and Version Control for Ada.* Carnegie-Mellon Univ., May, 1980.

[Habermann82]
Habermann, Nico. *The Second Compendium of Gandalf Documentation.* Carnegie-Mellon Univ., May, 1982.

[Harslem82]
Harslem, Eric. A Retrospective on the Development of Star. *Proc. 6th International Conference on Software Engineering.* Tokyo, Japan, Sept., 1982.

[Heckel78]
Heckel, P. A Technique for Isolating Differences Between Files. *Comm. ACM 21,* 6 (April 1978).

[Horsley79]
Horsley, Thomas. Pilot: A Software Engineering Case Study. *Proc. 4th International Conference on Software Engineering.* 1979, 94.

[Ichbiah79]
Ichbiah, Jean D. Preliminary ADA Reference Manual. *SIGPLAN Notices 14,* 6 Part A (June 1979).

[IEEE]
IEEE. *Standard Glossary of Software Engineering Terminology.* IEEE Standard 729-1983, IEEE.

[Ince84]
Ince, D.C. A Source Code Control System Based on Semantic Nets. *Software Practice and Experience 14,* 12 (Dec. 1984), 1159-1168.

[Ivie77]
Ivie, E. The Programmer's Workbench - a Machine for Software Development. *Comm. ACM 20,* 10 (Oct. 1977), 746.

[Kaiser82]
Kaiser, G., and Nico Habermann. *A Description of the Correct Version Control Supported by the Gandalf Environment.* Carnegie-Mellon Univ., 1982.

[Katz85]
Katz, R.H., M. Anwaruddin, and E. Chang. *A Version Server for Computer-Aided Design Data.* UCB/CSD 86/266, U.C. Berkeley, Nov., 1985.

[Lampson83a]
Lampson, Butler W., and Eric Schimdt. Practical Use of a Polymorphic Applicative Language. *Proc. 10th POPL Conference.* ACM, June, 1983.

[Lampson83b]
Lampson, Butler W., and Eric Schmidt. Organizing Software In a Distributed Environment. *SIGPLAN Notices 18,* 6 (June 1983).

[Lauer79]
Lauer, Hugh, and Ed Satterthwaite. Impact of Mesa on System Design. *Proc. 4th International Conference on Software Engineering.* IEEE, Sept., 1979, 174-182.

[Leblang84]
Leblang, David. Computer Aided Software Engineering in a Distributed Environment. *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* ACM, May, 1984, 104.

[Leblang85]
Leblang, David. Configuration Management for Large-Scale Software Development Efforts. *Proc. Workshop on Software Engineering Environments for Programming-in-the-Large.* GTE Laboratories, June, 1985, 122.

[Lewis83]
Lewis, Brian. Experience with a System for Controlling Software Versions in a Distributed Environ-

ment. *Proc. Symposium on Application and Assessment of Automated Tools for Software Development.* IEEE, Nov., 1983. IEEE Press catalog number 83CH1936-4.

[Lewis84]
Lewis, Brian. *IncludeChecker,* in Xerox Development Environment Users' Guide Ed. , Xerox Information System Division, 1984.

[Linton84]
Linton, Mark. Implementing Relational Views of Programs. *Proc. SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* ACM, May, 1984, 132.

[Liskov77]
Liskov, Barbara H., Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction Mechanisms in CLU. *Comm. ACM 20,* 8 (Aug. 1977), 564.

[Minsky84]
Minsky, N. The Darwin Software-Evolution Environment. *Proc. SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* ACM, May, 1984.

[Mitchell79]
Mitchell, James G., William Maybury, and Richard Sweet. *The Mesa Language Manual.* CSL-79-3, Xerox PARC, April, 1979.

[Nicklin]
Nicklin, Peter. MKMF - Makefile Editor. In *UNIX Programmer's Manual 4.2 BSD.*.

[Nicklin83]
Nicklin, Peter. *The SPMS Software Project Management System.* UC Berkeley, Aug., 1983.

[Parnas72a]
Parnas, David L. *Use of the Concept of Transparency in the Design of Hierarchically Structured Systems.* Carnegie-Mellon Univ., 1972.

[Parnas72b]
Parnas, David L. On the Criteria To Be Used In Decomposing Systems into Modules. *Comm. ACM 15,* 12 (Dec. 1972), 1053.

[Powell82]
Powell, Michael, and Mark Linton. *The OMEGA Programming System.* , 1982.

[Powell83]
Powell, Michael, and Mark Linton. A Database Model of Debugging. *Proc. ACM SIGSOFT/ SIGPLAN Software Engineering Symposium on High Level Debugging.* ACM, March, 1983, 365-375.

[Prager83]
Prager, J.M. The Project Automated Librarian. *IBM Syst. J.* 22, 3 (1983), 214.

[Reiss84]
Reiss, Stephen P. Graphical Program Development with PECAN Program Development Systems. *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* May, 1984, 30.

[Rochkind75]
Rochkind, Marc. The Source Code Control System. *IEEE Trans. Software Eng. 1,* 4 (Dec. 1975), 364-370.

[Rowland83]
Rowland, B.R. Software Development System. *Bell Systems Tech. J. 62,* 1 (Jan. 1983).

[Schmidt82]
Schmidt, Eric E. *Controlling Large Software Development In a Distributed Environment.* CSL-82-7, Xerox PARC, Dec., 1982.

[Teitelbaum75]
Teitelbaum, Warren. *The INTERLISP Reference Manual.* Xerox PARC, 1975.

[Teitelbaum81a]
Teitelbaum, Warren. The INTERLISP Programming Environment. *IEEE Computer 14,* 4 (April 1981).

[Teitelbaum81b]
Teitelbaum, Tim. The Cornell Program Synthesizer: A Syntax Directed Programming Environment. *Comm. ACM 24* (Sept. 1981), 563.

[Teitelbaum83]
Teitelbaum, Warren. Cedar: An Interactive Programming Environment for a Compiler Oriented Language. *Proc. LALN/LLNL Conference on Work Stations in Support of Large Scale Computing.* March, 1983.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS NONE			
2a. SECURITY CLASSIFICATION AUTHORITY N/A		3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED			
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) SEI-SM-4-1.0		5. MONITORING ORGANIZATION REPORT NUMBER(S)			
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST.	6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE			
6c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213		7b. ADDRESS (City, State and ZIP Code) ESD/AVS HANSCOM AIR FORCE BASE HANSCOM, MA 01731			
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE	8b. OFFICE SYMBOL (If applicable) ESD/ AVS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962890C0003			
8c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213		10 SOURCE OF FUNDING NOS.			
		PROGRAM ELEMENT NO. 63752F	PROJECT NO. N/A	TASK NO. N/A	WORK UNIT NO. N/A

11. TITLE (Include Security Classification)
Software Configuration Management

12. PERSONAL AUTHOR(S)
James E. Tomayko, The Wichita State University

13a. TYPE OF REPORT FINAL	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) September 1986	15. PAGE COUNT 66

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	configuration management
			change control
			software evolution

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

These materials support the curriculum module SEI-CM-4-1.3 "Software Configuration Management."

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☒	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED DISTRIBUTION	
22a. NAME OF RESPONSIBLE INDIVIDUAL JOHN S. HERMAN, Capt, USAF	22b. TELEPHONE NUMBER (Include Area Code) 412 268-7630	22c. OFFICE SYMBOL SEI JPO

Curriculum Modules (* Support Materials available)

CM-1   [superseded by CM-19]
CM-2   Introduction to Software Design
CM-3   The Software Technical Review Process*
CM-4   Software Configuration Management*
CM-5   Information Protection
CM-6   Software Safety
CM-7   Assurance of Software Quality
CM-8   Formal Specification of Software*
CM-9   Unit Testing and Analysis
CM-10   Models of Software Evolution: Life Cycle and Process
CM-11   Software Specifications: A Framework
CM-12   Software Metrics
CM-13   Introduction to Software Verification and Validation
CM-14   Intellectual Property Protection for Software
CM-15   Software Development and Licensing Contracts
CM-16   Software Development Using VDM
CM-17   User Interface Development*
CM-18   [superseded by CM-23]
CM-19   Software Requirements
CM-20   Formal Verification of Programs
CM-21   Software Project Management
CM-22   Software Design Methods for Real-Time Systems*
CM-23   Technical Writing for Software Engineers
CM-24   Concepts of Concurrent Programming
CM-25   Language and System Support for Concurrent Programming*
CM-26   Understanding Program Dependencies

Educational Materials

EM-1   Software Maintenance Exercises for a Software Engineering Project Course
EM-2   APSE Interactive Monitor: An Artifact for Software Engineering Education
EM-3   Reading Computer Programs: Instructor's Guide and Exercises